## DOCTOR OF ENGINEERING (ENGD)

# Programming Models for Heterogeneous Systems with Application to Computer Graphics

Potter, Ralph

*Award date:*
2017

*Awarding institution:*
University of Bath

# Programming Models for Heterogeneous Systems with Application to Computer Graphics

submitted by

Ralph Potter

for the degree of Doctor of Engineering

of the

University of Bath

Department of Computer Science

June 2017

## ABSTRACT

For over a decade, we have seen a plateauing of CPU clock rates, primarily due to power and thermal constraints. To counter these problems, processor architects have turned to both multi-core and heterogeneous processors.

Whilst the use of heterogeneous processors provides a route to reducing energy consumption, this comes at the cost of increased complexity for software developers. In this thesis, we explore the development of C++-based programming models and frameworks which enable the efficient use of these heterogeneous platforms, and the application of these programming models to problems from the field of visual computing.

Two recent specifications for heterogeneous computing: SYCL and Heterogeneous System Architecture, share the common goal of providing a foundation for developing heterogeneous programming models. In this thesis, we provide early evaluations of the suitability of these two new platforms as foundations for building higher-level domain-specific abstractions. We drawing upon two use cases from the field of visual computing: image processing and ray tracing; and explore the development and use of domain-specific C++ abstractions layered upon these platforms.

We present a domain-specific language which generates optimized image processing kernels by deeply embedding within SYCL. By combining simple primitives into more complex kernels, we are able to eliminate intermediate memory accesses and improved performance.

We also describe Offload for HSA: a single-source C++14 compiler and programming model for Heterogeneous System Architecture. The pervasive shared virtual memory offered by HSA allows us to reduce execution overheads and relax constraints imposed by SYCL's programming model, leading to significant performance improvements.

Performance optimization on heterogeneous systems is a challenging task. We build upon Offload to provide RTKit, a framework for exploring the optimization space of ray tracing algorithms on heterogeneous systems.

Finally, we conclude by discussing challenges raised by our work and open problems that must be resolved in order to unify C++ and heterogeneous computing.

# ACKNOWLEDGEMENTS

I would firstly like to express my sincere gratitude to both my academic supervisor, Dr. Russell Bradford, and to my industrial supervisors, Dr. Alastair Murray and Dr. Paul Keir. All three have provided invaluable support, guidance and insight throughout the duration of my EngD studies.

Furthermore, I wish to extend my thanks to Codeplay Software for their sponsorship of my studies; for providing a pleasant and stimulating environment in which to conduct my research; and for their kindness and understanding in granting me the time to bring my work to completion. I feel extremely grateful to have been given the opportunity to work amongst such a unique set of skilled and knowledgeable engineers.

Particular thanks are due to Dr. Jens-Uwe Dolinsky for his continual aid and insights on all matters compiler-related, and for his continual willingness to answer my endless questions.

I also wish to thank the Centre for Digital Entertainment for providing the support and equipment that I have needed to complete my thesis, and the Engineering and Physical Sciences Research Council for providing both the opportunity and the funding for my studies.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF LISTINGS

# Part I

# Introduction & Background

# 1 INTRODUCTION

## 1.1 MOTIVATION

Since the early 1970's, ongoing reductions in logic gate sizes have provided continuous improvements in Central Processing Unit (CPU) clock frequencies. In 1965, Gordon Moore initially observed that the density of transistors in integrated circuits doubled every year, later revising the observation to every two years. This trend came to be known as Moore's Law (Mollick, 2006). For software limited by arithmetic throughput, these improvements in clock frequencies resulted in commensurate reductions in execution times. However, processor architects reached the limit of this scaling around 2005, primarily due to the interrelated problems of energy consumption, heat dissipation and leakage voltages at very small process sizes (Märtin, 2014).

At this time, CPU architects turned to multi-core CPU designs. Sutter (2005) famously declared "The Free Lunch Is Over", and that software developers would need to embrace parallelism and concurrency. Multi-core designs allowed CPUs to scale within energy budgets through added parallelism, at the cost of increased software complexity.

More recently, we have seen a rise in heterogeneous systems, which aim to deliver improved performance through the use of multiple specialized Processing Units (PUs) with differing architectural and performance characteristics. This improved performance may manifest as throughput or latency improvements, or in terms of improved power efficiency. These systems have been adopted across a wide spectrum of use cases, from supercomputers to smartphones. Heterogeneous systems are able to offer further reductions in energy consumption and improvements in throughput due to their potential to match workloads to PUs optimized for the specific characteristics of the processing tasks. However, this introduces further complexity for software developers. In addition to the challenges of parallelism and concurrency introduced by multi-core CPUs, developers targeting heterogeneous systems must also address the

architectural differences between PUs, matching algorithms to the most appropriate PU.

Many heterogeneous systems also feature complex segmented memory subsystems, the efficient use of which is key to improving performance and reducing energy usage. These memory systems can offer improved performance by providing better data locality relative to a particular PU, or through the use of memory hierarchy designs that are well matched to the characteristics and typical workload of a particular PU. However, managing this data locality imposes further burdens on application developers.

With this architectural shift towards heterogeneous designs comes a need for new programming models to address the rising complexity of systems. Higher-level languages and abstractions can aid in addressing this rising complexity. In this thesis, we aim to produce domain-specific programming models and frameworks which enable the efficient use of these heterogeneous platforms.

This thesis is primarily concerned with the development of C++-based programming models and libraries which build upon two recent open standards for heterogeneous computing: SYCL (Khronos OpenCL Working Group – SYCL subgroup, 2015) and Heterogeneous System Architecture (HSA) (HSA Foundation, 2015a,b,c). These two standards share the common goal of providing foundations which higher-level domain-specific languages and libraries can utilize to access the resources of heterogeneous systems, albeit with significant differences in approach.

SYCL provides a high-level C++ abstraction layer and single-source programming model, targeted at application and library developers and building upon OpenCL (Khronos OpenCL Working Group, 2012) in order to enable hardware acceleration on heterogeneous processors. By contrast, HSA targets a much lower level of abstraction, aiming to provide the necessary foundational primitives to enable the development of new parallel languages and runtimes for heterogeneous systems.

In both cases, the work described in this thesis was conducted concurrent to the development of each specification. The work was able to provide valuable early feedback based on practical experience whilst both specifications were under development. This helped inform both my own and Codeplay Software's input into the specification of each standard.

## 1.2 OUTLINE OF THIS THESIS

Over the course of this thesis, we will explore two approaches to applying heterogeneous systems to problems from the field of visual computing.

Over the course of this thesis we will describe three technical works:

- An approach to building a Domain Specific Language (DSL) for image processing on heterogeneous devices, based upon SYCL.
- Offload: A C++ compiler and programming model for HSA, providing more advanced capabilities than SYCL.
- RTKit: a C++-based ray tracing framework for exploring performance optimization on heterogeneous systems, based on Offload.

These works share a number of common themes:

- Providing early usage experience and validation to two new standards for heterogeneous computing: SYCL and HSA
- The development and application of new C++ programming models for heterogeneous computing.
- The use of those programming models to build domain-specific toolkits for problems in the field of visual computing.

This thesis touches on aspects drawn from a number of diverse fields including computer hardware, compilers, programming models, and aspects of image processing and ray tracing from the field of computer graphics. As an aid to the reader, Chapter 2 provides a review of the necessary foundational aspects of these fields. Due to a relative lack of existing work focusing on programming with either SYCL or HSA, an introduction to both platforms is also included.

Each of these fields are active research fields in their own right. Chapter 3 provides a review of related work. This chapter will primarily cover programming models for heterogeneous systems, DSLs for parallel image processing, kernel fusion and ray tracing on heterogeneous systems.

Chapter 4 focuses on building a DSL for image processing. The high cost of memory accesses can have a negative impact on application performance on massively parallel accelerators such as Graphics Processing Units (GPUs). Chapter 4 describes an approach to implementing embedded domain-specific languages on SYCL in order

to generate optimized kernels with minimal intermediate memory accesses. The constraints imposed by SYCL's programming model introduce some specific challenges which need to be addressed. The work described in this chapter ultimately proved generalizable to a number of additional libraries and situations on SYCL.

The challenges described in chapter 4 are ultimately rooted in the OpenCL 1.2 memory model (Khronos OpenCL Working Group, 2012), and the model that SYCL uses to abstract that. Chapter 5 describes a C++ programming model which resolves many of these issues by building on Heterogeneous System Architecture. We describe an extended single-source C++ compiler with support for HSA's complex segmented memory model, and a runtime library to manage communication, synchronization and work scheduling within a heterogeneous system.

Efficiently mapping workloads to a new heterogeneous system is a complex task. Algorithmic performance may vary considerably based on the suitability of the hosting processor, and the cost of data movement between processors can be significant. Furthermore, the need to rewrite source code into different languages or dialects in order to evaluate potential optimizations discourages experimentation. In chapter 6 we explore RTKit, a C++-based ray tracing framework which builds upon our single-source programming model described in chapter 5 to explore the performance characteristics of ray tracing on HSA and Accelerated Processing Units (APUs).

Finally, we make some concluding remarks in chapter 7. We discuss the implications and limitations of our work, describe potential future avenues for research and explore the challenges and unresolved questions that must be overcome in order to unify ISO C++ and heterogeneous computing.

## 1.3 CONTRIBUTIONS OF THIS THESIS

This thesis makes the following contributions:

Image processing is an important use-case for heterogeneous processors. Providing libraries of image processing primitives, implemented as OpenCL kernels, can provide application developers with easy access to hardware acceleration, without requiring machine expertise. However, implementing individual operators as single OpenCL kernels can lead to increased memory bandwidth overheads, when compared to composing multiple operators together to form larger kernels. Previous approaches to applying kernel fusion to image processing on OpenCL devices have either required external tools, or the implementation of considerable compilation machinery. We explore whether SYCL's single-source programming model is sufficiently expressive to enable the generation of fused kernels, without requiring further external tools. We present an embedded domain-specific language for image processing which utilizes the SYCL standard to provide acceleration on OpenCL devices. We demonstrate how such an embedded language can generate fused GPU kernels in order to eliminate intermediate memory accesses. We evaluate the performance of these fused kernels, demonstrating improved performance over image processing pipelines implemented in OpenCV and Halide.

Whilst the approach described above can reasonably be applied to a subset of image processing operators, it is not generalizable to a wide range of problem domains. The Heterogeneous System Architecture (HSA) specifications are intended to provide a foundation for the development of heterogeneous programming models and runtimes.

In a similar spirit to our exploration of the applicability of SYCL to image processing, we explore the suitability of HSA to accelerating ray tracing. HSA's pervasive use of shared virtual memory, along with lightweight work dispatch and synchronization, potentially allow for the fine-grained distribution of tasks between multiple heterogeneous processors.

Due to focusing on providing foundational primitives for the construction of parallel programming models and runtimes, HSA lacked any high-level language suitable for application development. In order to resolve this lack of suitable languages, and to facilitate the exploration of the performance characteristics of HSA, we present two works: Offload, a C++14-based compiler and programming model for HSA; and

RTKit, a ray-tracing framework designed to aid in the exploration of the optimization space on heterogeneous systems.

We introduce a new C++14-based programming model for Heterogeneous System Architecture. Through our single-source C++14 compiler and runtime library, we provide an early example of such a work. We discuss the compiler implementation and associated runtime library facilitating our programming model, and provide comparisons highlighting the advantages of our model over existing programming models such as CUDA and C++ AMP.

Optimization on heterogeneous systems is a complex task. We describe RTKit, a framework to aid in exploring the optimization space for ray tracing applications on heterogeneous architectures. We are able to exploit the ease of code migration, and the pervasive shared virtual memory found in our heterogeneous C++ programming model to explore a number of different algorithmic permutations and hardware mappings and evaluate their performance.

Additionally, all three of the works described above serve to provide validation of the SYCL and HSA specifications. These works were all developed prior to, and concurrent with, the publication of the corresponding specifications. Beyond addressing the specific challenges addressed above, each of these projects also provided an opportunity to evaluate the usability and suitability of each specification.

## 1.4 IMPACT

The work described in this thesis has both lead to academic publications, and helped to guide and inform input to industry standards.

### 1.4.1 Publications

Portions of the work contained in this thesis have previously been described in the following publications:

Potter, R., Bradford, R.J., Murray, A. and Dolinsky, U., 2016. A C++ programming model for Heterogeneous System Architecture. In: M. Taufer, B. Mohr and J.M. Kunkel, eds. *High performance computing - ISC high performance 2016 international*

*workshops, revised selected papers* [Online], Frankfurt, Germany. Vol. 9945, Lecture Notes in Computer Science, pp.433–450. Available from: http://dx.doi.org/10.1007/978-3-319-46079-6_31.

Potter, R., Keir, P., Bradford, R.J. and Murray, A., 2015. Kernel composition in SYCL. In: S. McIntosh-Smith and B. Bergen, eds. *Proceedings of the 3rd international workshop on OpenCL, IWOCL 2015* [Online], 13–13 May 2015 Palo Alto, California, USA. ACM, 11:1–11:7. Available from: http://dx.doi.org/10.1145/2791321.2791332.

### 1.4.2 Standards and Specifications

The work described was conducted concurrent to the development of both the SYCL and HSA specifications. As such, it provided valuable practical feedback and experience. It has informed contributions to the following industry standards documents, both in terms of direct personal contributions and indirectly via colleagues:

HSA Foundation, 2015. *HSA runtime programmers reference manual* [Online]. (V.1.0). Available from: http://www.hsafoundation.com/?ddownload=4946.

Khronos OpenCL Working Group – SYCL subgroup, 2015. *SYCL specification* [Online]. (V.1.2). Khronos Group. Available from: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf.

These contributions include the relaxation of type constraints in SYCL kernel code, facilitating simplified sharing of complex data structures between host and accelerator code; and input on the representation of coarse and fine-grained memory regions in HSA.

### 1.4.3 Dissemination to Academic and Industrial Audiences

The work described in this thesis has informed presentations to a number of academic and industrial audiences. A complete list can be found in appendix A.

Our work has resulted in several invited presentations to standards body working groups, aimed at informing the future design of specifications produced by the respective groups.

Potter, R., 2015. *Models for high level languages on HSA*. HSA Foundation Working Group. Cambridge, UK.

Potter, R., 2016a. *A C++ programming model for Heterogeneous System Architecture*. HSA Foundation Working Group. Edinburgh, UK.

Potter, R., 2016b. *A C++ programming model for Heterogeneous System Architecture*. SG14 - ISO C++ Study Group on Games Development and Low-Latency.

Potter, R., 2016c. *Generating efficient GPU kernels using expression templates*. HSA Foundation Working Group. Edinburgh, UK.

Knowledge gained throughout this thesis has also informed portions of several presentations given on behalf of standards organisations. These presentations are primarily focused on the outputs of the standards bodies themselves.

Potter, R., 2014a. *SPIR 2.0 provisional*. OpenCL BoF, SIGGRAPH. Vancouver, Canada.

Potter, R., 2014b. *SYCL: A system which integrates C++ with OpenCL*. UK Many-Core Developer Conference. Cambridge, UK.

Potter, R., 2015. *SYCL overview*. Khronos Group - OpenCL, SYCL & SPIR-V, SIGGRAPH. Los Angeles, US.

Potter, R., 2016. *SPIR-V: A shader IR for OpenCL, OpenGL and Vulkan*. Khronos Group - OpenCL, SYCL & SPIR-V, SIGGRAPH. Anaheim, USA.

Our work has also lead to presentations to academic, industrial and open-source community audiences, primarily discussing technical details and implementation approaches to C++ on accelerators.

Lomüller, V., Potter, R. and Dolinsky, U., 2016. *C++ on accelerators: supporting single-source SYCL and HSA programming models using clang*. European LLVM Developers' Meeting. Barcelona, Spain.

Potter, R., 2015. *Implementing khronos SYCL for OpenCL*. LPGPU Workshop on Power-Efficient GPU and Many-core Computing. Amsterdam, The Netherlands.

Potter, R., 2016. *A C++ programming model for Heterogeneous System Architecture*. UK Many-Core Developer Conference. Edinburgh, UK.

Potter, R., Keir, P., Lucas, J., Alvarez-Mesa, M., Juurlink, B. and Richards, A., 2013. *Fusing GPU kernels within a novel single-source C++ API* [Online]. HiPEAC Compiler, Architecture and Tools Conference. Haifa, Israel. Available from: https://software.intel.com/en-us/articles/compiler-architecture-and-tools-conference-2013-abstract.

## 1.5 A BRIEF NOTE ON TERMINOLOGY

This thesis primarily focuses on two specifications for heterogeneous computing, SYCL and HSA. Whilst we will focus on these two specifications, throughout this thesis references will also be made to the functionality and behaviour found in other specifications, most notably OpenCL (Khronos OpenCL Working Group, 2012), CUDA (NVIDIA Corporation, 2007) and C++ AMP (Microsoft Corporation, 2013). Regrettably, these specifications each choose to adopt their own nomenclature for common features. In order to provide a common frame of reference, we will adopt the terminology from HSA throughout this thesis.

# 2 BACKGROUND

This thesis will touch on topics from several disciplines within computer science. As an aid to the reader, this chapter aims to provide a foundational understanding of the relevant topics and concepts. This will be followed in chapter 3 by a deeper examination of select published works.

We will begin in section 2.1 with a brief discussion of the historical trends and engineering challenges that have led to the rise of heterogeneous systems in recent years.

This is followed by an introduction to heterogeneous systems themselves in section 2.2. This section aims to furnish the reader with a basic understanding of heterogeneous systems, as well as illustrating their ubiquity. This section primarily focuses on the hardware characteristics of heterogeneous systems.

The rise of heterogeneous systems has lead to a proliferation of programming models and Application Programming Interfaces (APIs) to address the challenges of programming these complex devices. Despite this proliferation, many of these models share common characteristics which we will discuss in general terms within this chapter. A more detailed review of the evolution of the software ecosystem for heterogeneous systems over the course of the last 15 years is reserved for chapter 3.

Because heterogeneous systems are composed of several Processing Units (PUs) with differing characteristics, programming models for heterogeneous systems must tackle the challenge of segmenting programs and targeting regions of code at individual PUs. In section 2.3 we discuss the various compilation models adopted by language runtimes for heterogeneous systems to address this.

Many heterogeneous runtimes share a common execution model, strongly influenced by the properties of modern Graphics Processing Unit (GPU) architectures. In section 2.4, we discuss this kernel execution model and its implications.

Heterogeneous devices often also feature complex memory systems, enabling improved data locality for specific PUs. Section 2.5 discusses the memory models found

in a variety of heterogeneous environments. This section primarily concerns the topics of coherency, and models for implicit versus explicit data movement.

This will conclude our discussion of the commonalities between programming models. We will then focus on two recent additions to the software ecosystem for heterogeneous systems: SYCL (Khronos OpenCL Working Group – SYCL subgroup, 2015) and Heterogeneous System Architecture (HSA) (HSA Foundation, 2015a,b,c). Chapter 4 focuses on our approach to implementing a Domain Specific Language (DSL) for image processing on top of SYCL, while chapter 5 describes the implementation of a new C++14-based single-source programming model targeting HSA. The work described in these two chapters was conducted concurrent to the development of each of the respective specifications and aided both my own and Codeplay Software's input into each specification. Due to the immaturity and restricted availability of these platforms, they are less well known than other platforms and are yet to achieve the same level of adoption as major frameworks such as CUDA (NVIDIA Corporation, 2007) and OpenCL (Khronos OpenCL Working Group, 2012). As an aid to readers who are less familiar with these platforms, we provide a brief introduction to the salient details of each platform. We begin with an introduction to SYCL in section 2.6, followed by an exploration of HSA in section 2.7.

The fields of image processing and ray tracing will provide motivating use-cases for our work. As previously discussed, chapter 4 describes our DSL for image processing on heterogeneous systems. Chapter 6 describes our work on the design and implementation of RTKit, a framework for ray tracing on heterogeneous systems. RTKit builds upon our previously mentioned C++14 compiler and runtime library for HSA. Therefore, the final sections of this chapter will introduce some key concepts from the fields of image processing (section 2.8) and ray tracing (section 2.9), and describe how these algorithmic concepts impact different types of PU in a heterogeneous system.

Finally, we will conclude the chapter with a brief recap in section 2.10.

## 2.1 MOTIVATION

The last decade has seen a rise in heterogeneous systems as Central Processing Unit (CPU) clock frequencies have plateaued, primarily due to power constraints. In 1974, Dennard et al. (1974) observed that as transistors reduced in size, power density

remains constant. This came to be known as Dennard scaling. Coupled with Moore's Law (Mollick, 2006), Dennard scaling resulted in exponential increases in processor clock frequencies for over 30 years. In conjunction with architectural improvements, these increases in clock frequency in turn led to continually improving performance for sequential code. Around 2005, Dennard scaling began to break down, primarily due to power and thermal constraints (Märtin, 2014).

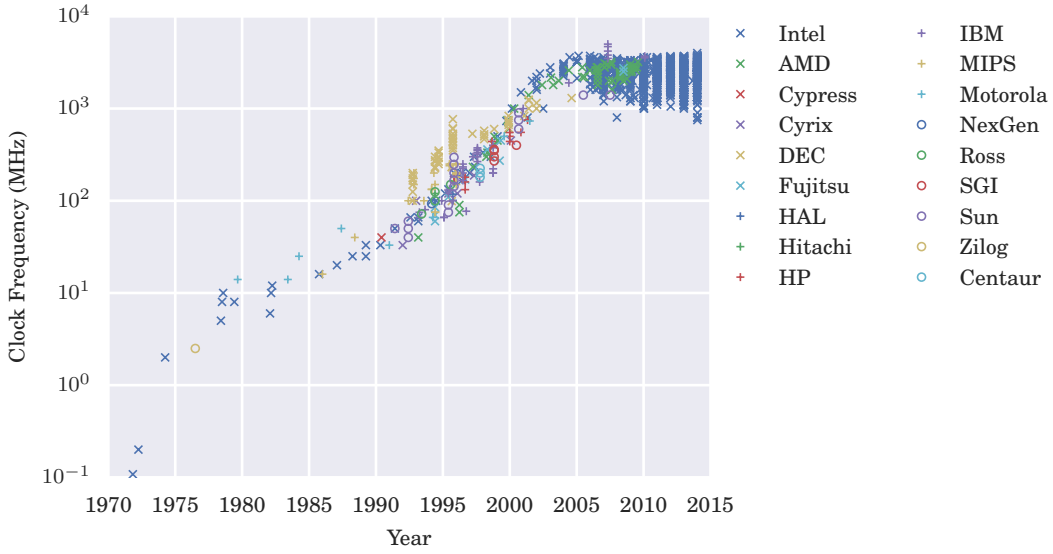



**Figure 2-1:** CPU Clock Frequencies - Data from cpu-db (Danowitz et al., 2012)

Figure 2-1 uses data from CPU DB (Danowitz et al., 2012) to visualise base clock frequencies for CPUs produced by a range of manufacturers since 1970. We can observe that CPU clock frequencies showed relatively consistent exponential scaling between 1975 and 2005. This is followed by either a plateauing or reduction in clock frequencies from 2005 forwards.

Unable to continue to scale single-core clock frequencies within reasonable power budgets, hardware architects turned first to multi-core and then later to heterogeneous designs.

Multi-core CPU designs feature multiple identical CPU cores, delivering increased performance through increased parallelism. By contrast, heterogeneous systems aim to deliver improved performance through the use of multiple specialized PUs with differing architectural and performance characteristics. The most common form of heterogeneous system is to couple one or more general-purpose PUs with specialized data-parallel PUs such as GPUs. Dependent on the specific system design, the per-

formance benefits of a heterogeneous system may manifest as throughput or latency improvements, or in terms of improved power efficiency.

For both multi-core and heterogeneous systems, efficiently exploiting the available performance of these hardware designs requires increased software complexity. The introduction of multi-core CPUs forced software developers to address concurrency and parallelism within their code in order to harness the performance of multiple cores. Concurrency control and the elimination of data races has become a necessary consideration when programming modern multi-core systems.

In the case of heterogeneous systems, the use of multiple forms of PU leads to further complexity for software developers. These systems retain the parallel nature of multi-core systems, but also introduce challenges relating to the architectural differences between PUs. To achieve optimal performance, computational tasks must now be targeted at the appropriate PU, based on the characteristics of specific computational algorithms; and on the architectural characteristics of the available PUs. Heterogeneous systems often also feature complex memory subsystems, the efficient use of which is key to improving performance and reducing energy usage (Che, Sheaffer and Skadron, 2011; L. Chen and Agrawal, 2012; Wuytack et al., 1994).

Some of this complexity can be alleviated through the use of high-level languages and programming models. In this thesis, we aim to produce programming models and frameworks which enable the efficient use of these heterogeneous platforms.

## 2.2 AN INTRODUCTION TO HETEROGENEOUS SYSTEMS

Heterogeneous computing involves the use of computing systems containing two or more kinds of PUs, typically with different Instruction Set Architectures (ISAs). Through the use of dissimilar PUs, and the mapping of tasks to the most suitable PU, significant power and efficiency gains can be made (Y. Wang and Cheng, 2012).

Heterogeneous designs have been used for a diverse range of hardware. At extreme scale, all of the top 40 supercomputers in the Green500 (Green 500, 2015) list use some combination of conventional CPUs paired with accelerator devices. In desktop PCs, a conventional CPU coupled with a discrete GPU can be considered a simple heterogeneous system. In the same space, the integration of CPU and GPU cores into a single chip has become widespread, with examples such as Advanced Micro

Devices (AMD)'s Llano (Branover, D. Foley and Steinman, 2012) and Intel's Sandy Bridge (Yuffe et al., 2011) and Ivy Bridge (Damaraju et al., 2012). At the opposite end of the spectrum, heterogeneous processors are also common in mobile System-on-Chip (SoC) processors. Here we see both single-ISA heterogeneity where large and small cores are used for power efficiency (Kumar et al., 2003), and mixed-ISA designs with CPU, GPU and Digital Signal Processor (DSP) cores (Qualcomm, 2017).

One common model is to designate one or more PUs as the primary processor(s). This PU is typically responsible for managing the overall state of the system, dispatching work to other PUs and performing tasks which other PUs lack the capabilities to perform such as resource allocation or I/O. We refer to this PU as the host processor, or more simply host. This role is typically filled by a conventional CPU.

Under this model, the remaining PUs are utilized as co-processors, performing the tasks assigned to them by the host processor. We refer to these PUs as accelerator devices.

Whilst some differences can be found in terms of hardware capabilities, terminology and programming models, there is still substantial commonality between the major General Purpose Graphics Processing Unit (GPGPU) frameworks. In the subsequent sections we will discuss compilation models (section 2.3), the kernel execution model (section 2.4) and memory models (section 2.5).

## 2.3 COMPILATION MODELS FOR HETEROGENEOUS SYSTEMS

In a heterogeneous system, we can expect to find two or more PUs with differing ISAs. Consequently, a compilation toolchain must be capable of identifying which regions of code are intended for execution on each processor and generating output in the appropriate instruction set. This may involve separate translation units for each PU, making identification and code segmentation trivial. Alternatively, some form of source code analysis may be required to perform the segmentation. Furthermore, whilst the ISA of the host processor is typically assumed to be known prior to execution, the presence of specific accelerator devices and their specific ISAs is often not known until runtime.

In order to resolve this lack of prior knowledge regarding the availability of accelerator devices, a common approach is to classify regions of code based on the PU

on which they will be executed, and partially delay compilation. Code intended to execute on the host processor can be compiled offline into the host processor's native ISA. Code intended to execute on accelerator devices can be stored in a more abstract form and compiled down to the ISA of a specific accelerator device at runtime. This abstract form might be the original human-readable source code such as OpenCL C (Khronos OpenCL Working Group, 2016) or GLSL (Khronos OpenGL Working Group, 2016b), or an intermediate language such as SPIR-V (Khronos SPIR-V Working Group, 2016) or HSAIL (HSA Foundation, 2015b).

In cases where high-level kernel or shader languages such as OpenCL C (Khronos OpenCL Working Group, 2016) or GLSL (Khronos OpenGL Working Group, 2016b) are consumed at runtime through online compilation, a compiler frontend must be embedded within the host runtime API implementation. This approach results in several trade-offs, based on the online nature of the compilation. As a runtime method, syntax errors now become runtime errors, rather than occurring offline at host compilation time. The inclusion of a compiler frontend within the runtime implementation adds also considerable complexity. Furthermore, online compilation is likely to be more time and resource sensitive than offline compilation. Consequently, long-running or resource-intensive compiler optimization passes which would be deemed acceptable in an offline compiler may be undesirable in an online compiler. On the positive side, online compilation also provides great flexibility for the dynamic generation and manipulation of kernels. Online compilation also resolves the problem of how to generate kernels in the native ISA of an accelerator device, where the target accelerator device is not known at host compile time.

As an alternative to consuming high-level languages at runtime, accelerator code can be compiled directly to the native ISAs of specific accelerator devices offline. This removes the need to include complex compiler infrastructure within a runtime implementation. However, it requires that the available accelerators be known a priori and results in a kernel representation which is not portable between accelerators. This approach can be suitable for applications where the target hardware is fixed, such as High Performance Computing (HPC). However, it is difficult to scale for applications required to run on more diverse hardware, such as smartphone or desktop applications.

A compromise between these two extremes is to use a two-stage compilation process. Offline compilation can be used to transform regions of accelerator code into a simpler, device-agnostic intermediate language such as Heterogeneous System Architecture Intermediate Language (HSAIL) (HSA Foundation, 2015b) or SPIR-V (Khronos

SPIR-V Working Group, 2016). This enables tasks traditionally performed in a compiler frontend such as syntax checking and semantic analysis to be performed offline. Device-agnostic code transformations and optimizations can also be applied at this stage, circumventing the time and resource constraints found in online compilation techniques.

At runtime, a compiler embedded in the runtime implementation can then transform this intermediate form into the native ISA of the target accelerator device. This process is referred to as finalization. Whilst this approach still requires the inclusion of a compiler in the runtime implementation, this compiler has much more limited scope. This approach retains the capacity to generate representations of kernel functions in the native ISA of accelerator devices at run-time, whilst retaining the majority of the benefits of offline compilation. In the field of general-purpose computing, an analogy can be drawn to the Java virtual machine, where applications are compiled to platform agnostic bytecode offline and this bytecode may in turn be Just In Time (JIT) compiled to a native instruction set at runtime. This model is adopted within HSA via HSAIL (HSA Foundation, 2015b); OpenCL 2.1 via SPIR-V (Khronos SPIR-V Working Group, 2016) and RenderScript (Hines et al., 2011) via LLVM Intermediate Representation (IR).

Orthogonal to the final representation of executable code is the process of classifying host and accelerator code, and the source languages used to represent each form. Several different variants exist here.

Dual-source programming models expect separate translation units for host and accelerator code, typically with different languages or dialects and compiler invocations. Kernels and shaders are typically expressed in specialized kernel or shading languages, separate from the language used to target the host processor. This approach allows for domain-specific specialization of the kernel or shading language. This comes at the cost of limiting code reuse and added complexity when migrating code between host and accelerator PUs. OpenCL (Khronos OpenCL Working Group, 2008) and OpenGL (Khronos OpenGL Working Group, 2016a) are examples of the dual-source model.

Single-source compilers allow for a single translation unit to be used for both host and accelerator code, typically with some form of annotation to aid the compiler in identifying which regions should map to a particular instruction set (Eichenberger et al., 2006; International Business Machines, 2008). This reduces the complexity of code migration and interoperability, simplifying the process of incremental porting

of code between PUs. CUDA (NVIDIA Corporation, 2007) and OpenMP (OpenMP ARB, 2015) are single-source models.

The SYCL specification defines a third variant: shared-source compilation (Khronos OpenCL Working Group – SYCL subgroup, 2015, p. 177). As with single-source, shared-source compilation utilizes a single common language for host and device compilation. However, two separate compiler invocations are used to generate host and device executables. This approach retains the ease of use and code portability found in single-source models, while allowing developers some freedom in their choice of host compiler. In order to provide this flexibility with respect to choice of host compiler, a programming model must avoid requiring non-standard language extensions within the source code. SYCL is expressed entirely in standard C++, and thus SYCL code can be parsed by any standards-compliant C++11 compiler.

A single-source or shared-source model can reasonably be implemented as an abstraction layer above a dual-source model through the use of source-to-source translation. For example, SYCL could be implemented by translating the C++ kernel code to OpenCL C offline, and then using the online compiler embedded within an OpenCL runtime to generate native code for target accelerator devices.

## 2.4 KERNEL EXECUTION MODEL

OpenCL, HSA and CUDA all provide low-level runtimes to support heterogeneous computation. These are discussed further in chapter 3. All three of these runtimes feature similar execution models, albeit with some small differences in nomenclature.

Under these three models, computation to be performed on accelerator devices is expressed in terms of kernel functions.

Each kernel dispatch launches a one, two or three-dimensional grid of light-weight software threads, each evaluating an instance of the same kernel function. These threads are referred to as work-items. Each work-item is assigned a unique coordinate within the grid, which can be utilized to index data or to aid in distributing work between work-items.

The grid of work-items is further subdivided into work-groups and wavefronts, as described below. Figure 2-2 provides an illustration of this hierarchical organisation of work-items into wavefronts, work-groups and a top-level grid.

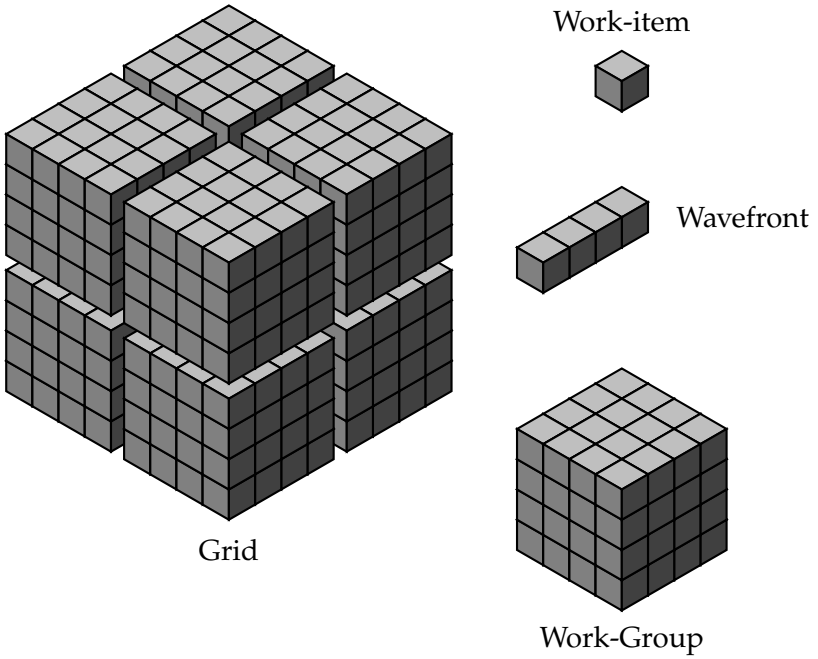



**Figure 2-2:** Kernel Execution Model

Each work-item has its own set of registers, has private memory, and can access a unique set of read-only values such as work-item indices through a set of special instructions.

On some platforms, work-items may be gang-scheduled into groups corresponding to the width of Single Instruction, Multiple Data (SIMD) execution units of the hosting accelerator device. OpenCL introduced the notion of a sub-group to encapsulate this concept, while HSA uses the term wavefront. Work-items within a wavefront cannot make independent forward progress due to sharing a program counter. However, wavefronts are able to independently make forward progress with respect to each other. Some runtimes, such as HSA, provide functionality to enable synchronization and operations such as broadcasting, shuffling and balloting between work-items within a single wavefront.

The grid is subdivided into fixed-size blocks of work-items referred to as work-groups. Work-items within a single work-group may share access to dedicated group storage such as the on-chip scratchpad memory as found in GPUs. Work-items within a single work-group may also synchronize execution through barrier operations. All work-items in a single work-group are guaranteed to be executed on a single compute unit. They may be executed concurrently, or through some form of scheduling. Whilst work-items within a single work-group execute concurrently and

are able to synchronize and communicate, multiple work-groups may be executed sequentially, in parallel, and in an unspecified order with respect to each other.

Consequently, there is no safe way to synchronize execution between work-groups within a single kernel dispatch, or for a work-group to wait on the results of computations by another work-group within the same kernel dispatch. Such synchronization can be accomplished by splitting a computation into two kernels. This constraint is particularly significant to our work on combining image processing operators into single kernels, as described in chapter 4, because it restricts the form of operators that can be safely combined.

## 2.5 MEMORY MODELS FOR HETEROGENEOUS SYSTEMS

In section 2.4, we described how work-items have access to both their own dedicated private memory, and to shared work-group storage. Figure 2-3 provides a more complete visualization of this relationship.

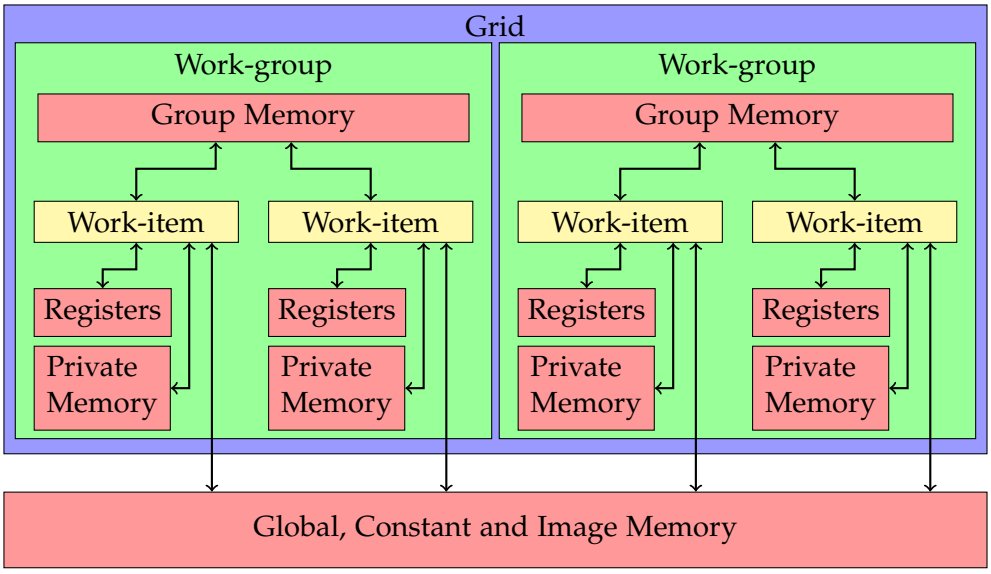



**Figure 2-3:** Kernel and Memory Model

From figure 2-3, we can observe multiple distinct representations of memory, with differing access restrictions. However, this diagram represents a purely conceptual model exposed by heterogeneous programming models, albeit one that closely mirrors the structure of modern GPUs.

The mapping of these conceptual representations of memory to physical memory within a PU is typically the responsibility of the runtime implementation of a given heterogeneous programming model.

Some heterogeneous systems are architected such that all PUs share access to the same physical memory. This model is common in smartphones (Akenine-Möller and Ström, 2008). However, in some cases accelerator devices may also feature their own dedicated memory, providing improved locality and reduced access costs from the associated accelerator device. This dedicated memory may offer reduced performance, or be wholly inaccessible, when accessed from other PUs.

For example, modern discrete GPUs provide small amounts of dedicated Static Random-Access Memory (SRAM) on-chip to provide extremely high bandwidth access to group memory; provide specialized data paths to constant and image memory, typically backed by Graphics Double Data Rate (GDDR) memory chips located on the same hardware board; and back global memory with either GDDR on-board or system memory accessed across a Peripheral Component Interconnect Express (PCIe) bus.

### 2.5.1 Memory Coherence and Consistency

One problem that arises with heterogeneous systems is that of memory coherence (Li and Hudak, 1986). Memory coherence requires that following a write operation on a given memory address, any subsequent read operations on the same address return the last value written. This property is challenging to maintain in a system where multiple PUs are simultaneously manipulating the same region of memory. Where coherence describes behaviour with respect to individual memory locations, memory consistency is concerned with the ordering of accesses to all memory locations. For a more thorough treatment of memory coherence and consistency models, we direct readers to Sorin, Hill and Wood (2011).

Memory consistency models are a complex topic, the finer details of which are largely beyond the scope of this thesis. However, a basic understanding is necessary in order to discuss the properties of programming models for heterogeneous systems. For a deeper understanding the design and application of memory consistency models to heterogeneous systems, we refer readers to the work of Gaster et al. (Gaster, D. Hower and Howes, 2015; D.R. Hower et al., 2014).

One approach to maintaining memory consistency between PUs is to define a programming model which constrains how PUs may be granted access to a region of memory. We can define a model where the only valid states are to allow zero or more PUs read-only access, or a single PU read-write access, to a region of memory. Under this model, it is only necessary to update caches or migrate data between different physical memories dedicated to specific PUs on transitions between states. This approach operates at the granularity of whole memory allocations or buffers and is referred to as coarse-grained coherence.

When applied to the kernel execution model described above, coarse-grained coherence requires that all kernels must complete before a state transition which grants or removes write access to a buffer.

An alternative approach is to define a coherence model on a much finer granularity. We can define a model which grants multiple PUs simultaneous write access to the same region of memory. However, for such a model to be useful we must still be able to ensure the correct ordering of read and write operations. This can be accomplished by using atomic operations, memory fences and execution barriers as synchronization primitives. These primitives both constrain compiler and hardware reordering of instructions, and provide the information necessary to enable cache-coherence protocols to maintain a consistent view of memory across multiple PUs. We refer to this as fine-grained coherence.

Fine-grained coherence allows us to utilize data structures and algorithms that are challenging to implement under a coarse-grained model, such as shared lock-free queues. However, fine-grained coherence is not without cost. The programming model is complex, and maintaining cache-coherency between multiple PUs requires dedicated hardware support.

OpenCL 1.2 and SYCL adopt a coarse-grained coherence model. By contrast HSA defaults to fine-grained coherence, but optionally supports the allocation of buffers which relax this to coarse-grained coherence.

### 2.5.2 Implicit vs. Explicit Data Movement

Where accelerator devices feature dedicated memory, it may be desirable or even necessary that data be moved from host memory into this dedicated memory before computations on that data are performed on the accelerator device, and results

may require moving back to host memory following the completion of a computation.

Programming models for heterogeneous systems need to address this requirement and enable the management of this data movement. This can either be expressed explicitly as operations which an application programmer must perform to move data, or it can be an implicit part of the programming model, with the language runtime assuming the responsibility.

Explicit models such as OpenCL and CUDA provide API functions for use by application developers to schedule this data movement. These explicit models can offer potential performance advantages by enabling the overlapping of computation on a host processor with asynchronous data transfers. However, this comes at the cost of additional complexity for application developers.

By contrast, implicit models such as C++ Accelerated Massive Parallelism (C++ AMP) and SYCL attempt to derive sufficient information to manage this data movement internally within the runtime library, without needing the application developer to explicitly manage data movement.

Whilst the memory model in early versions of CUDA was based solely on explicit data movement, CUDA 6 introduced support for runtime-managed unified memory. This provides developers with optional access to a simplified programming model, albeit it with an associated performance cost in many cases (Landaverde et al., 2014).

This concludes our discussion of the conceptual compilation, execution and memory models found in heterogeneous systems. In the next two sections we will discuss the two specifications for targeting heterogeneous systems: SYCL and HSA.

## 2.6 OPENCL AND SYCL

OpenCL is an open standard for parallel computation on heterogeneous systems, designed and specified by the Khronos Group. OpenCL defines a runtime API with C and C++ bindings (Khronos OpenCL Working Group, 2008, 2013) and separate kernel languages: OpenCL C (Khronos OpenCL Working Group, 2016) and OpenCL C++ (Khronos OpenCL Working Group, 2017). These kernel languages are dialects derived from C99 and C++11 respectively. Incompatibilities between these kernel dialects and their corresponding base languages present a barrier to code reuse between

host and device code. Additionally, OpenCL defines a kernel execution model which corresponds to the execution model described in section 2.4.

The OpenCL specifications have evolved through several iterations. Until OpenCL 2.0, the host and accelerator devices used disjoint address spaces and data movement between these address spaces required the use of explicit memory copy API functions. OpenCL 2.0 introduced support for several variations on Shared Virtual Memory (SVM), and the optional use of a single shared address space.

Also designed by the Khronos Group, SYCL is a C++11-based cross-platform parallel programming abstraction layer built on top of OpenCL. OpenCL utilizes a dual-source compilation model, with a separate kernel language. By contrast, SYCL provides a shared-source model, where code for both the host processor and accelerator devices are written entirely in standard C++ and can be intermingled in a single source file. This aims to relax the restrictions on code reuse and ease porting of existing C++ code to accelerators.

SYCL addresses several of OpenCL's weaknesses with respect to interoperability with C++. OpenCL applications typically require considerable boilerplate code, much of which is abstracted into a terser and more convenient form in SYCL. Through shared-source compilation, and by sharing a common language between host and accelerator code, SYCL provides strong type safety across the host-accelerator boundary.

This type safety, coupled with support for the use of C++ templates as kernel functions, brings us to a key strength of SYCL. For library developers, replicating the functionality provided by C++ templates in OpenCL not well supported. A number of OpenCL libraries attempt to provide a set of kernels parameterized by data type by utilizing the C preprocessor to generate multiple variants of kernels. For example, OpenCV (Itseez, 2015) takes this approach to provide support for images with differing bit-widths or data types. Libraries requiring the generation of more complex kernels are forced to turn to dynamic generation of kernels through string manipulation. VexCL (Demidov, 2012) provides an example of this. VexCL is primarily a linear algebra library, but is forced to implement considerable compilation machinery within the library to accomplish kernel generation. These challenges are not solely limited to OpenCL C. Whilst the recently released OpenCL C++ kernel language (Khronos OpenCL Working Group, 2017) does support the use of C++ templates internally within a kernel, OpenCL C++ prohibits the use of templates as kernel functions themselves. Consequently, OpenCL C++ fails to adequately address this challenge.

We will rely heavily on SYCL's support for templated kernel functions in our work on generating kernels from an embedded DSL in chapter 4.

In contrast to OpenCL's model of explicit data movement, SYCL's programming model is designed to provide the runtime library with sufficient information to transparently manage implicit data movement. SYCL combines wrapper types for OpenCL concepts such as buffers, images and events with a scheduler to automatically manage data transfers between devices.

The inclusion of SVM in OpenCL 2.0 might appear to alleviate the need to provide a framework such as SYCL's for scheduling data movement. In practice, the specification of SVM in OpenCL 2.0 suffers from a number of limitations in this respect. Hardware support for OpenCL 2.0 is much more limited than OpenCL 1.2, due in part to more demanding requirements. Additionally, the OpenCL 2.0 specifications define multiple different variants of SVM, with varying allocation and synchronization requirements. Only one of these variants (coarse-grained buffer SVM) is a core requirement for all OpenCL 2.0 implementers, with the remaining variants being optional features. Whilst coarse-grained buffer SVM does ease the sharing of data structures between PUs by ensuring that a single common address space is used, its use still requires granting a single PU exclusive access to the buffer through the use of map and unmap operations. As a result, some mechanism for scheduling the access is still required.

Kernels in SYCL are written using a restricted subset of C++11. Kernels are implemented as standard C++ function objects in the same translation unit as host code, with the function call operator providing the kernel body. This allows for the reuse of functions between the host and device code. This differs from OpenCL, where kernels are typically expressed in OpenCL C and code reuse is complicated by language differences between OpenCL C and C99. The SYCL function objects may either be traditional named class types, or the anonymous unnamed closure types generated by lambda expressions.

```
1 kernel void vector_add(global float* a, global float* b,
2                         global float* c) {
3   int i = get_global_id(0);
4   a[i] = b[i] + c[i];
5 }
```

**Listing 2-1:** A Vector Addition Kernel in OpenCL C 1.2

Listing 2-1 illustrates a kernel written in OpenCL C 1.2, while listing 2-2 shows a functionally equivalent SYCL kernel. Due to the structure of the respective programming models, listing 2-2 illustrates both a kernel function and related code to launch execution, whilst listing 2-1 illustrates only the kernel function. The requirement for explicit address space keywords in OpenCL C 1.2, as seen in listing 2-1 is relaxed in OpenCL 2.0, at the cost of introducing additional hardware requirements. By design, SYCL 1.2 can be implemented purely as a software layer above an existing OpenCL 1.2 implementation.

```
cgh.parallel_for<class vector_add>(cl::sycl::range<1>(1024),
  [=](cl::sycl::id<1> id) {
    a[id] = b[id] + c[id];
  }
);
```

**Listing 2-2:** A Vector Addition Kernel in SYCL

SYCL is unable to rely upon hardware features beyond the minimum functionality required by OpenCL 1.2. Therefore, a number of C++ features are not supported within SYCL kernels, or any function called from a SYCL kernel. Function pointers are prohibited. This extends to functionality which is dependent on support for function pointers, such as virtual function calls, and runtime type information. Exceptions, dynamic memory allocation and runtime recursion within kernels are also prohibited.

Outside the scope of kernels, the limitations of OpenCL 1.2 introduce a number of constraints into the design of SYCL. OpenCL 1.2 lacks support for SVM and allows for the use of separate address spaces by the host processor and accelerator devices. As a result, pointers may only be dereferenced on their associated device. Furthermore, due to both the implicit data movement in SYCL's programming model and the constraints of OpenCL's memory model, device addresses cannot be assumed to be persistent between separate kernel executions. Functionality which requires a common address space between the host processor and the accelerator devices is prohibited. Most notably, this prevents the use of mutable global variables.

Unlike OpenCL, compilation in SYCL is an offline process. SYCL introduces the notion of a device compiler, a compilation step responsible for identifying regions of code intended to be executed on an OpenCL device and generating a representation suitable for execution. This may be accomplished through the use of separate compilers for the host system and accelerator devices, or through the use of a single integrated compiler. In the case of Codeplay's SYCL implementation (Codeplay Software

Ltd., 2016), this compilation step generates Standard Portable Intermediate Representation (SPIR) 1.2 bitcode, which is embedded in the final executable.

### 2.6.1 Buffers and Accessors

As described previously, SYCL provides a transparent memory model with implicit data movement. This model is constructed on top of the OpenCL 1.2 model of explicit memory copy APIs. In order to perform efficiently, a SYCL runtime must be able to infer when data must be moved between accelerator devices; moved to or from host memory; and when data movement can be avoided entirely.

In order to do this, SYCL uses a system of buffers and accessors. This is broadly similar to, and derives from, work by Howes et al. (2009a).

A SYCL buffer is an opaque object with an associated dimensionality and internal element type. The contents of a buffer cannot be accessed directly, and the physical location of the data stored within a buffer is entirely managed by the SYCL runtime. Over the course of the lifetime of a buffer, a SYCL runtime might reasonably migrate the contents of buffer between host system memory and dedicated memory located on one or more accelerator devices.

Whilst a buffer in SYCL performs a similar role to an OpenCL buffer, there are some differences. In OpenCL, a buffer is represented by a `cl_mem` handle. A `cl_mem` object is bound to an OpenCL context, and consequently to a set of devices belonging to a single OpenCL platform. A buffer in SYCL is not constrained in the same manner. On a system with multiple installed OpenCL platforms, a buffer may be required to migrate between two devices which do not share a context, such as between an Intel CPU and a discrete GPU. Therefore, there is no one-to-one mapping between a SYCL buffer and an OpenCL `cl_mem` object. Instead, a SYCL buffer may create and destroy many OpenCL `cl_mem` objects over the course of its lifetime.

The contents of buffers cannot be accessed and manipulated directly. Instead, accessor objects are required. Accessors serve a dual purpose in SYCL. They provide a view onto the data held in a buffer, with convenient and type-safe array syntax. They also provide a method by which the access requirements of kernels can be codified. For example, an accessor must declare whether read-only or write-only access to the corresponding buffer is required. This information can then be used by the SYCL

runtime to construct a data flow graph and so efficiently schedule kernel execution and data movement.

### 2.6.2 Scopes

As described previously, code within a SYCL kernel function is constrained to a restricted subset of C++, due to the limitations of OpenCL 1.2 hardware implementations. However, these constraints do not apply to code which will execute solely on the host processor. We can think of the kernel function as forming a scope, a region of program execution in which a set of constraints apply.

We can view SYCL applications as a nested set of three scopes. Kernel scope forms the innermost scope, and applies to the body of the function object call operator which represents kernel in SYCL. Code within this scope may be executed on an accelerator device, and so must adhere to the language constraints previously described i.e. no use of functions pointers, or dereferencing of host memory addresses.

Our next scope is command group scope. Enqueuing a kernel to be executed in SYCL is accomplished by passing an instance of a C++ lambda or function object to the SYCL queue `submit` function. This is illustrated in listing 2-3. This function object will be executed on the host processor, and so the restrictions seen in kernel scope do not apply here.

```
1  // Host scope
2  queue.submit([&](cl::sycl::handler& cgh) {
3    // Command group scope.
4    using access = cl::sycl::access::mode;
5
6    auto device_a = buffer_a.get_access<access::write>(cgh);
7    auto device_b = buffer_b.get_access<access::read>(cgh);
8    auto device_c = buffer_c.get_access<access::read>(cgh);
9
10   cgh.parallel_for<class vector_add>(cl::sycl::range<1>(1024),
11     [=](cl::sycl::id<1> id) {
12       // Kernel function scope.
13       device_a[id] = device_b[id] + device_c[id];
14     }
15   );
16 });
```

**Listing 2-3:** Host, Command Group and Kernel Scopes in SYCL.

Command group scope is the only scope in which we can construct SYCL accessors. This scope is the only one in which `cl::sycl::handler` objects are accessible to developers. These handler objects are necessary in order to create the `cl::sycl::accessor` objects that are used within kernels to provide access to data. Accessors are transient objects, which can be viewed as forming the edges within a data flow graph. This data flow graph will in turn be used by the SYCL runtime to schedule data movement and kernel executions. These restrictions on the scope in which accessors can be constructed will directly impact the implementation of our DSL later in chapter 4.

Our outermost and final scope is host or application scope. Code at this scope is executed on the host processor, and standard C++ rules apply without additional constraints. SYCL buffers and images may only be constructed at application scope.

### 2.6.3 Implementations

The SYCL specification was published in April 2015. At the time of writing there are no fully conformant implementations publicly available.

Two open-source implementations are currently under development: triSYCL (Keryell, 2015), a CPU-only implementation based on OpenMP; and SYCL-GTX (Žužek, 2016), where an embedded DSL is used to generate OpenCL C kernels at runtime.

A third, proprietary, implementation is under development by Codeplay Software: ComputeCpp (Codeplay Software Ltd., 2016). The ComputeCpp implementation is used as the underlying SYCL implementation for our image processing DSL in chapter 4 and for comparative benchmarks to our C++ programming model for HSA in chapter 5.

### 2.6.4 Summary

SYCL provides a shared-source C++11-based abstraction layer built upon OpenCL. It brings modern C++ features such as templates and lambda functions to OpenCL 1.2 hardware and provides a memory model based on implicit data movement.

One of the primary design goals of SYCL is to provide a foundation which C++-based libraries and frameworks can utilize to access the computational power of the

```
// Create a queue to work on
cl::sycl::queue queue;

// Create and initialize input vectors.
float a = 99.0f
float *x = new float[N];
float *y = new float[N];
float *z = new float[N];
...

// Create some 1D SYCL buffers, wrapping our vectors.
cl::sycl::buffer<float, 1> buffer_x({N}, x);
cl::sycl::buffer<float, 1> buffer_y({N}, y);
cl::sycl::buffer<float, 1> buffer_z({N}, z);

// Asynchronously launch a command-group.
// This is a compute kernel combined with an associated access
// specification.
queue.submit([&](cl::sycl::handler& cgh) {
  using access = cl::sycl::access::mode;

  // We need read-only access to buffers x and y.
  auto access_x = buffer_x.get_access<access::read>(cgh);
  auto access_y = buffer_y.get_access<access::read>(cgh);

  // We need write access to z, discarding any existing contents.
  auto access_z = buffer_z.get_access<access::discard_write>(cgh);

  // Enqueue a parallel kernel with a 1D iteration space of N
  // work-items.
  cgh.parallel_for<class saxpy>({N}, [=](cl::sycl::id<1> index) {
    // The kernel lambda function captures the accessors and the
    // scalar a by-value, passing them as arguments to an
    // underlaying OpenCL kernel.
    access_z[index] = access_x[index] * a + access_y[index];
  });
});
```

**Listing 2-4:** Single-Precision $A \cdot X + Y$ implemented in SYCL.

wide range of accelerator devices that provide support for OpenCL. We provide an example of this use case in chapter 4, where we make use of template metaprogramming to implement a DSL for image processing. Without the functionality provided by SYCL, it would have been necessary to implement considerable compilation and scheduling functionality within our DSL's supporting runtime library.

SYCL is strongly focused on providing hardware acceleration to C++-based libraries and frameworks. In the next section, we will explore HSA, which shares the common goal of providing foundations for accessing the computing power of heterogeneous accelerators. However, HSA takes a much lower-level approach, providing specifications aimed at the designers of parallel languages and compilers, rather than C++ library developers.

## 2.7 HETEROGENEOUS SYSTEM ARCHITECTURE

HSA is a set of standards defining hardware capabilities (HSA Foundation, 2015a), runtime programming interfaces (HSA Foundation, 2015c) and a virtual instruction set (HSA Foundation, 2015b) to enable multiple processors or devices to communicate and interoperate through a shared memory system.

HSA is intended to provide a foundation on which higher-level models for parallel computation on heterogeneous systems can be built. Whilst this goal is similar to that of SYCL, HSA aims to provide a foundation for a wide range of languages and runtimes. As such, HSA offers a lower-level interface than other runtimes for targeting heterogeneous systems such as OpenCL. For example, HSA lacks a high-level kernel language such as OpenCL C, has no built-in maths library and more directly exposes the properties of memory and cache subsystems.

### 2.7.1 Agents

An HSA system consists of one or more agents. Agents are devices which participate in the HSA memory model.

Most commonly agents represent programmable devices such as CPUs, GPUs or DSPs. Alternatively, they may be fixed-function devices such as cameras or video en-

code/decode units; or software constructs exposed by the HSA runtime implementation such as a simulated PU designed for debugging or profiling purposes.

Regardless of architecture, agents present a common communication interface to other agents within a HSA system. They can all send and receive requests for the execution of work through Architected Queuing Language (AQL) packets (section 2.7.2), and communicate through signals or shared regions of cache-coherent memory. By providing this common interface, multiple PU are able to interact and communicate without the need to address the unique architectural details of each PU individually.

### 2.7.2 Queues and Architected Queuing Language

Agents receive requests to perform work from other agents in the system via user-mode queues. Each agent has an associated packet processor which will monitor user-mode queues, consume packets from them and schedule the execution of the requested task. Each user-mode queue is tightly bound to a single agent. Multiple user-mode queues may be created per agent, and multiple agents may submit requests to a single user-mode queue.

A user-mode queue consists of a block of user-space memory that is globally accessible within the system, a pair of atomic counters to enable the queue to be utilised as a ring buffer, and a signal to allow agents be woken from a low-power state when work is requested.

Agents submit requests to user-mode queues in the form of AQL packets. This submission process consists of manipulating the atomic counters to reserve space within the ring buffer, storing the packet itself at the associated memory address and updating the queue's signal to inform the packet processor of the corresponding agent that a new request may be available.

These simple memory operations can be performed by agents without the intervention of the host processor, enabling non-host agents to dispatch work to each other directly. Furthermore, because this process only involves the modification of user-space memory, it can be performed without requiring the intervention of the operating system kernel. This results in reduced latency when dispatching work between agents.

```
// We begin by atomically reserving the next available write
// position. Queues are effectively ring buffers, but read/write
// indices are monotonically increasing integers.
uint64_t write_id = hsa_queue_add_write_index_release(queue, 1);

// Create a signal to indicate when the kernel has completed
// execution. Initially set to 1. Will be set to 0 after the kernel
// completes execution
hsa_signal_t completion_signal;
hsa_signal_create(1, 0, nullptr, &completion_signal);

// Wait until the queue is not full before writing the packet.
uint64_t read_id;
do {
  read_id = hsa_queue_load_read_index_acquire(queue);
} while (write_id - read_id >= queue->size);

// Compute the packet location for the write index, considering
// wrap-around.
auto base = reinterpret_cast<hsa_kernel_dispatch_packet_t *>(
                                              queue->base_address);
auto packet = base + (write_id % queue->size);

// This function is user code to populate the kernel dispatch packet
// structure with the state needed to execute a kernel.
initialize_kernel_dispatch_packet(packet, info.range,
  kernel.private_size(),
  kernel.group_size() + info.dynamic_group_mem_size,
  kernel.address(),
  reinterpret_cast<uint64_t>(kernel_args),
  completion_signal);

// Switch the packet header state from HSA_PACKET_TYPE_INVALID
// to HSA_PACKET_TYPE_KERNEL_DISPATCH, so that the packet processor
// knows we are finally ready to execute.
__atomic_store_n(reinterpret_cast<uint8_t *>(&packet->header),
             static_cast<uint8_t>(HSA_PACKET_TYPE_KERNEL_DISPATCH),
             __ATOMIC_RELEASE);

// Signal the queue doorbell to wake the packet processor if it is
// powered down.
hsa_signal_store_release(queue->doorbell_signal,
                         static_cast<hsa_signal_value_t>(write_id));
```

**Listing 2-5**: Implementing Kernel Dispatch for HSA

Unlike the model exposed by other APIs such as OpenCL or CUDA, user-mode queues must be explicitly implemented and managed by the application developer. Listing 2-5 shows one possible implementation of this.

Four packet types are formally defined: kernel dispatch, agent dispatch, barrier-AND and barrier-OR. Further allowance is made for vendor or implementation defined packet types.

Kernel dispatch packets are used to request the execution of a kernel function over an N-dimensional work grid. We describe agents which support consuming these packets as kernel agents.

Agent dispatch packets provide a method to invoke functionality on an agent which does not support the kernel execution model. This might be a fixed function operation implemented in hardware, such as reading from a camera or decoding a video bitstream. Alternatively, it may be a conventional software function, implemented in the native ISA of the associated agent.

Vendor-specific packets allow for triggering implementation-specific behaviour.

Support for agent dispatch, kernel and vendor-specific packets is optional, although an agent must be able to support at least one of these in order to be capable of performing useful work.

All agents must also consume barrier-AND and barrier-OR packets. These packets delay the execution of subsequent packets until their dependencies are satisfied and enable the construction of task dependency graphs, while still allowing for the efficient execution of independent kernel or agent dispatch packets.

OpenCL 2.0 allows for device-side enqueue, where a kernel executing on an accelerator device may enqueue additional work to the same accelerator without requiring the intervention of the host processor. HSA's approach is more flexible, allowing any agent to enqueue work to both itself or to any other agent in the system.

### 2.7.3 Profiles and Machine Models

HSA has been designed to support a wide range of hardware. This ranges from large HPC compute nodes with multiple CPUs and discrete GPUs down to smartphones where multiple CPU, GPU and DSP cores maybe be packaged into a single SoC.

At the time of writing, HSA implementations are available for both discrete and integrated GPUs produced by AMD. In the mobile and embedded space, ARM, Imagination Technologies and MediaTek have all publicly discussed plans for future HSA-compliant hardware, but are yet to release implementations.

To accommodate this range of hardware, HSA defines hardware profiles and machine models. The HSA specifications define two profiles: base and full.

Base profile agents are only required to provide fine-grained coherency on buffers allocated using the HSA runtime API, and not system allocators such as `malloc`. Additionally, full profile agents are required to provide kernel preemption, status bits for detecting arithmetic exceptions and more comprehensive support for rounding modes in floating point arithmetic.

HSA also defines two machine models. The *small* machine model only addresses 4 GB of memory, making it suitable for embedded devices. Under this model, all pointers are represented as 32 bits. Under the *large* machine model some addresses remain 32 bits, while others are promoted to 64 bits. For example, addresses which are not accessible outside a single work-item or work-group remain 32 bits, while those which are globally accessible are increased in size under the large machine model.

### 2.7.4 Memory Model

Agents within an HSA system access shared system memory through a unified virtual address space. This unified address space ensures that a pointer passed between agents will remain valid, subject to some constraints that will be discussed later in this section. Enabling agents to exchange data by passing pointers, rather than requiring copies, can greatly reduce memory bandwidth requirements when transferring work between agents when compared to models such as OpenCL 1.2, which lacks similar guarantees.

The unified virtual address space is subdivided into logical segments. HSA defines seven segments which have differing allocation lifetimes, addressability, access rights and visibility of updates. These segments are disjoint regions of the virtual address space. The one exception to this is the kernarg segment. Kernel arguments appear within the kernarg segment during the execution of a kernel function, but are allocated from within a region of the global segment when dispatching a kernel.

| Segment | Granularity | Lifetime | Kernel Access | Host Access | Flat Address |
|---|---|---|---|---|---|
| Private | Work-Item | Work-Item | Read/Write | None | Yes |
| Group | Work-Group | Work-Group | Read/Write | None | Yes |
| Global | System/Agent | Application | Read/Write | Read/Write | Yes |
| Read-Only | Agent | Application | Read | Host API | No |
| Kernarg | Grid | Grid | Read | Write | No |
| Arg | Work-Item | Arg Block | Read/Write | None | No |
| Spill | Work-Item | Work-Item | Read/Write | None | No |

**Table 2-1:** Characteristics of Heterogeneous System Architecture Memory Segments

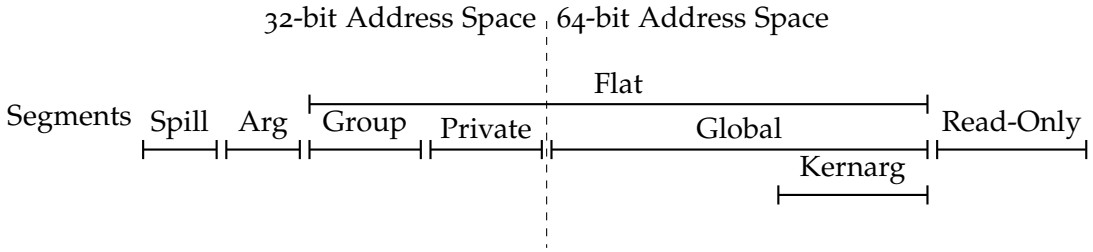



**Figure 2-4:** Example Segment Layout for HSA's Large Machine Model

The HSA specifications do not mandate the specific ordering and layout of these segments within the virtual address space. However, they do mandate the address sizes for each segment. This limits the set of potential layouts, with the positioning segments constrained by their corresponding address sizes. Figure 2-4 illustrates one possible partitioning of the 64-bit address space into segments.

The private segment holds variables that are local to a single work-item. The private segments for each work-item are overlaid on top of each other. This ensures that the physical storage associated with an address in the private segment can only be accessed by the work-item with which it is associated.

The group segment is used to hold variables shared by work-items within a single work-group. The address of a variable in the group segment can be read from or written to by any work-item within the associated work-group, but cannot be accessed by work-items outside that work-group, or by other agents within the system.

The lifetimes of allocations in the private and group segments are restricted to that of the associated work-items or work-groups. The global segment represents shared system memory and is used to hold data that persists beyond the duration of a single kernel dispatch. At least one sub-region of the global segment is accessible

to all agents in the system, including the host processor. Other sub-regions may be reserved for specific agents, or restricted subsets of agents.

The read-only segment can be used to hold variables that remain constant during the duration of a kernel execution.

The kernarg segment holds kernel arguments, and is read-only only from within a kernel dispatch. Values in the kernarg segment can be regarded as uniform for all work-items within a single kernel dispatch. Addresses within the kernarg segment are inaccessible from outside their associated kernel dispatch.

The arg segment is used to pass arguments to and from functions. Unlike the kernarg segment, variables in the arg segment are non-uniform across work-items and are only visible from the work-item with which they are associated.

HSA's virtual instruction set, HSAIL (HSA Foundation, 2015b), defines a finite number of virtual registers. Where the register budget is exceeded, a high-level compiler may choose to allocate values in the spill segment. This provides a hint to the finalizer that these values may be good candidates for promotion to hardware registers if additional registers are available during finalization. HSAIL itself is detailed further in section 2.7.5.

Instructions which transfer data between registers and memory, such as loads, stores and atomic instructions, encode the segment of their operands within the instruction. For example, `ld_global_u32` is a load of a 32-bit integer from an address in the global segment. We can see this in listing 2-7, where loading the kernel arguments from the kernarg segment into registers requires the use of `ld_kernarg` instructions, while loading the floating point operands for the addition from the global segment requires the use of `ld_global`.

Addresses may be associated with a particular segment, or they may be flat addresses. A flat address can be considered an address in a virtual segment that encompasses the private, group and global segments. Whilst flat addresses provide us with the convenience of using a single pointer type to address multiple segments, they are insufficient to completely free us from the need to track segments. Firstly, flat addresses cannot address the read-only segment. A model that operates entirely on flat addresses would therefore have to choose to eliminate the use of the read-only segment. This elimination has potential negative performance implications. Secondly, when dealing with HSA's large machine model, flat addresses are always 64 bits, regardless of which segment the address refers to. By contrast, private segment and group segment addresses are defined as 32 bits, regardless of memory model size. If

we consider an architecture such as Advanced Micro Devices (AMD) Graphics Core Next GPU (Advanced Micro Devices, 2016a), the number of vector registers used in a kernel constrains the number of wavefronts that may be active simultaneously and therefore the GPU's ability to hide the latency of memory accesses. Larger pointers require a larger portion of the available register budget, and so can negatively impact performance. Finally, using explicitly specified segment addresses provides the HSAIL finalizer with additional information which may enable further optimization or more efficient scheduling.

A heterogeneous system may contain multiple different forms of addressable memory. These may only be accessible to a subset of agents within a system, and have varying performance and coherence properties. A typical example of this is the dedicated DRAM found on many discrete GPUs. The HSA runtime API introduces the concept of a memory region. A region defines a further subdivision of a memory segment and represents a range of the virtual address space with a certain set of coherence and performance properties and accessible to some subset of the agents in the system.

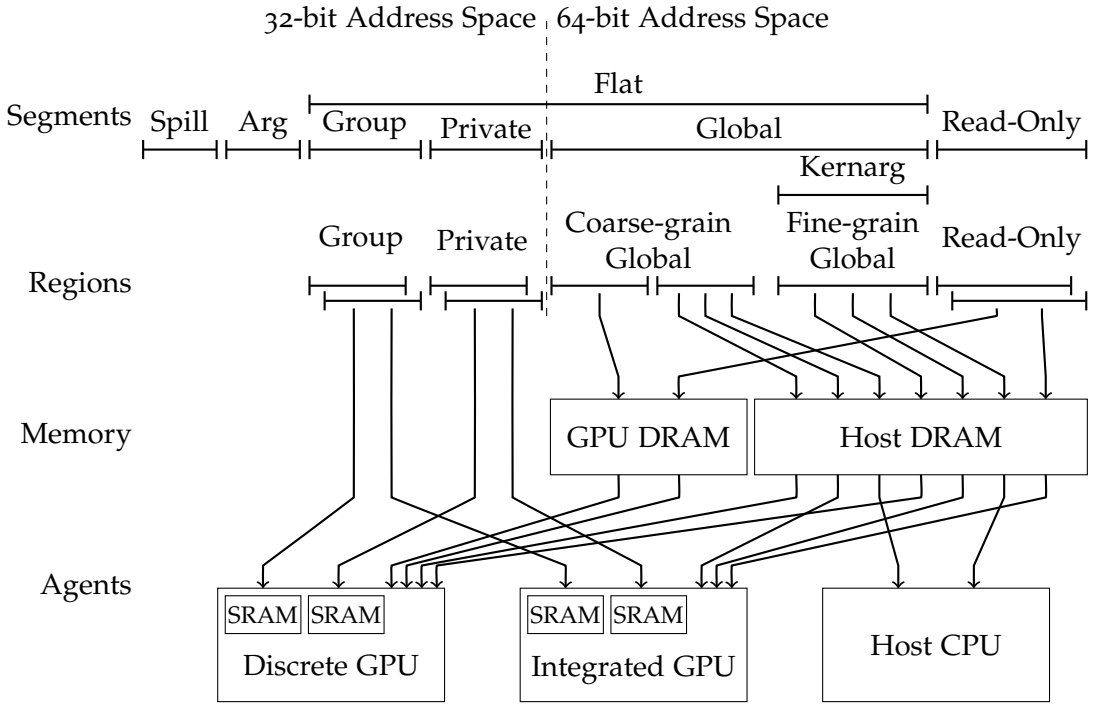



**Figure 2-5:** Example Mapping of HSA's Segments and Regions to Physical Memory

Figure 2-5 illustrates one possible mapping of segments and regions in a system with three agents. In this example, regions within the read-only, private and group segments map to different physical storage for each agent. Addresses within these segments may be safely aliased in this manner due to the access restrictions on these segments. By contrast, regions within the global segment may potentially be shared by multiple agents. In this example, we see a region with fine-grained coherency accessible to all agents. A second region with coarse-grained coherency is defined which is also accessible to all agents. Finally, a third region is accessible only to the discrete GPU.

HSA's architecture is based around multiple agents performing concurrent manipulation of shared memory. This necessitates a memory consistency model and coherence protocols. Maintaining system-wide coherence is potentially costly in terms of both latency and power.

The visibility of writes to shared virtual memory allocations in HSA is modified by a property referred to as granularity. HSA does not guarantee a consistent view of memory for every load or store instruction. Instead, the HSA memory model guarantees that each work-item or agent receives a consistent view with respect to a set of synchronization points. Memory allocated through the use of system allocators such as `malloc` or `new` is fine-grained. For these allocations, memory fences, atomic or signal operations, and kernel boundaries may all act as synchronization points. To further reduce the potential overheads of maintaining coherency, memory fences and atomic operations may optionally be augmented with scoping information. This allows the scoping of coherence to a single wavefront, work-group or agent, or system-wide. Some regions of the virtual address space may only support coarse-grained allocations. Memory from these regions is allocated through the `hsa_memory_allocate` API function. At any point in time, only a single agent may hold ownership of a coarse-grained allocation, and ownership is transferred via the `hsa_assign_agent` API function. HSA guarantees that the virtual address of a coarse-grained allocation remains constant when ownership is transferred between agents. However, the physical location backing an allocation may change in this case.

### 2.7.5 HSAIL and BRIG

HSA does not specify a high-level programming language such as OpenCL C. Instead, it defines a virtual instruction set, enabling compiler developers to target their

choice of high-level language at an HSA system. This virtual instruction set is called Heterogeneous System Architecture Intermediate Language (HSAIL) (HSA Foundation, 2015b). HSAIL also has a binary representation, Brigantine, the HSAIL binary format (BRIG) (HSA Foundation, 2015b, p. 302). For the remainder of the thesis, we treat the term HSAIL as encompassing both the textual and binary forms.

HSA relies upon a split compilation model. Kernels intended for execution on an HSA implementation are typically compiled from a high-level language such as C or C++ to HSAIL, either offline or through just-in-time (JIT) compilation. The HSAIL output from this compilation stage is device-agnostic and not natively executable on HSA agents.

In order to execute a kernel on an HSA agent, the HSAIL representation must be finalized into the native instruction set of the agent on which the kernel will be executed. This translation from HSAIL to native ISA is the responsibility of the HSA runtime implementation.

Listing 2-6 illustrates a simple vector addition expressed in OpenCL C, while listing 2-7 provides a translation into HSAIL.

```
1 kernel void vector_add(global float* a, global float* b,
2                         global float* c) {
3   int i = get_global_id(0);
4   a[i] = b[i] + c[i];
5 }
```

**Listing 2-6:** A Vector Addition Kernel in OpenCL C 1.2

The use of an intermediate language within HSA has a number of advantages given HSA's target audience. HSA is intended as a low-level platform upon which parallel programming languages and runtimes can be built, rather than as a platform for application developers.

An intermediate language provides a common target suitable for a variety of third-party compilers to emit code for. This allows third-party compilers to focus on front-end tasks such as semantic analysis, and provides a degree of portability by freeing each compiler from having to emit code in multiple agent-specific ISAs. Whilst a higher-level language similar to OpenCL C could also be targeted in this manner, a higher-level language offers few advantages as a compiler output format beyond provided limited readability for the purpose of debugging.

```
prog kernel &vector_add(kernarg_u64 %a,
                        kernarg_u64 %b,
                        kernarg_u64 %c)
{
  // Load the value of pointers a, b, c.
  ld_kernarg_align(8)_width(all)_u64  $d1, [%a];
  ld_kernarg_align(8)_width(all)_u64  $d2, [%b];
  ld_kernarg_align(8)_width(all)_u64  $d3, [%c];
  // Get the 1D work-item index in the grid.
  workitemabsid_u32 $s0, 0;
  // Convert index to byte-offset.
  cvt_u64_u32 $d0, $s0;
  shl_u64 $d0, $d0, 2;
  // Add offset to base pointers.
  add_u64 $d1, $d1, $d0;
  add_u64 $d2, $d2, $d0;
  add_u64 $d3, $d3, $d0;
  // Load floating point inputs.
  ld_global_align(4)_f32  $s0, [$d2];
  ld_global_align(4)_f32  $s1, [$d3];
  // Add inputs.
  add_ftz_f32 $s0, $s1, $s0;
  // Store the result.
  st_global_align(4)_f32  $s0, [$d1];
  ret;
};
```

**Listing 2-7**: A Vector Addition Kernel in HSAIL

Intermediate languages are also typically relatively simple to emit from a compiler, possessing simple, consistent syntax.

Intermediate languages require a comparatively simple compiler front-end, which in turn leads to a smaller and simpler runtime library implementation, and a reduced risk of inconsistencies between different hardware vendors implementation of the runtime library.

### 2.7.6 Summary

Where SYCL provides a framework focused on enabling parallelism for C++11-based applications and libraries, HSA aims to provide lower-level primitives to enable

language and compiler developers to provide support for heterogeneous computing.

HSA defines a hardware architecture based on one or more host CPUs, along with additional specialized PUs, communicating through a cache-coherent SVM system. Additionally, HSA specifies a small runtime library; and a virtual instruction set, HSAIL.

In chapter 5, we will make use of these tools to implement a C++14-based programming model. The pervasive cache-coherent SVM found in HSA will allow for much more fine-grained sharing of data structures and concurrency than is currently possible in existing models such as SYCL.

## 2.8 FUNDAMENTALS OF IMAGE PROCESSING

In this section, we will provide a brief introduction to image processing. Chapter 4 describes our work on using SYCL to implement a DSL which generates OpenCL kernels from primitive operators. Image processing provides the real-world use case for this work, and the algorithmic characteristics of image processing operators directly impact our implementation approach.

We will provide a brief introduction to image processing operators, discuss some of their properties and how they interact with the kernel execution model previously described in section 2.4. This section is not intended to provide a comprehensive review of image processing techniques. Rather, it is intended to illustrate some of the algorithmic characteristics that we find in image processing pipelines, and how these may map to a massively parallel processor such as a GPU.

Image processing comprises a set of techniques which take one or more images as inputs, and generate either new images or a set of characteristics relating to the input images as outputs.

One way to characterise image processing algorithms is as a series of one or more operators chained together to form a pipeline. These operators can be classified into three groups based on the manner in which they sample their input operands, and map to an output value.

Point operators are those that sample a single coordinate in an input image, and generate a single output value at the corresponding coordinate in an output image. We

can see an example of this mapping in figure 2-6. Examples of point operators include simple arithmetic operators such as addition, or colour space conversion operators. These are the simplest operators to map onto the kernel execution model previously described. A naive approach is to map each discrete coordinate in the input domain to a single work-item in the work grid of a kernel dispatch. Composing multiple point operators within a single kernel function is also trivial, simply requiring sequential execution of the operators in order of application, without any modification of work-group or grid layouts or intra-work-item synchronization.

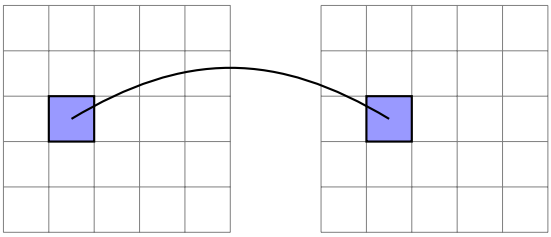


**Figure 2-6:** Example of a Point-wise Image Processing Operator

Local operators sample inputs from some local neighbourhood surrounding a coordinate in order to calculate an output value. Figure 2-7 illustrates this, with a nine pixel input region being sampled to generate a single output pixel. The classic Sobel operator (Sobel and Feldman, 1968) is an example of a local operator, computing the vertical and horizontal derivatives at each coordinate in an image by sampling the eight neighbouring coordinates.

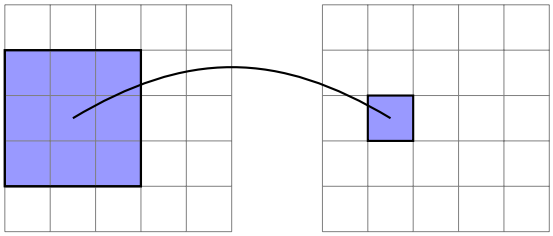


**Figure 2-7:** Example of a Local Image Processing Operator

Because local operators sample within some radius around each coordinate, we must define the expected behaviour for calculating the output value coordinates located close to the boundaries of the image. For these coordinates, sampling a neighbourhood may result in sampling from coordinates that fall outside the bounds of the input image. Possible strategies include returning a constant value such as zero, mirroring or repeating the image, or simply shrinking the size of the output image such that samples always fall within the bounds of the input image.

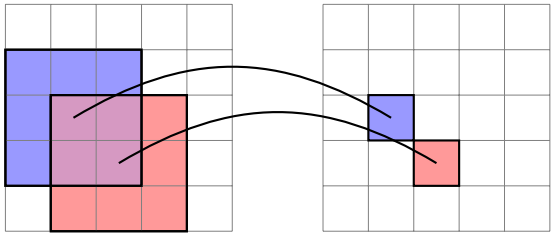


**Figure 2-8:** Overlapping Local Image Processing Operators

When attempting to evaluate local operators belonging to neighbouring output pixels, there may be substantial overlap between the sampled input regions, as shown in figure 2-8. This has implications for how we might attempt to chain operators. Consider a simple pipeline composed of a point operator $P$ to an input image $f$, followed by a local operator $L$ to the result. This is the functional composition: $L \circ P \circ f$. A more concrete example of such a pipeline might be converting a colour image to greyscale, and then applying a Sobel operator to detect edges in the resulting greyscale image.

In mapping this example pipeline to the kernel execution model, several approaches are possible here. We might evaluate the two operators individually as separate kernel functions, storing the output of $P$ in global memory before evaluating $L$. This minimizes the number of times $P$ is evaluated, at the cost of memory bandwidth.

Alternatively, we might choose to have each instance of $L$ independently re-evaluate $P$ across the local neighbourhood, *rematerializing* (Briggs, K.D. Cooper and Torczon, 1992) the output of $P$. This reduces the total memory bandwidth required, at the cost of repeated computation.

Finally, we might attempt to exploit tiling, and group memory which is likely to be located on-chip, to share the result of evaluating $P$ between work-items within a work-group. Under this approach, we evaluate $P$ and store the result in group memory. When we evaluate $L$, we can now retrieve the output of $P$ from group memory, rather than recomputing or performing a costly load from global memory. Whilst this approach is potentially attractive, group memory is often a scarce resource, requires additional synchronization between work-items, and requires special tile boundary handling logic due to the lack of synchronization between work-groups.

Global operators must sample entire input image in order to generate a single output. Consider the discrete Fourier transform:

$$F(x,y) = \frac{1}{NM} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} f(u,v) e^{-2\pi i (\frac{xu}{N} + \frac{yv}{M})} \tag{1}$$

where: $x = 0 \ldots N - 1$
$y = 0 \ldots M - 1$

In order to calculate the value of a single output pixel, $F(x,y)$, we must sample $f(u,v)$ for all values of $u$ and $v$.

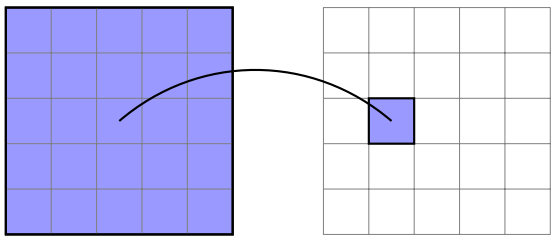


**Figure 2-9:** Example of a Global Image Processing Operator

A global operator can be considered analogous to a local operator where the neighbourhood encompasses the entire image. However, the scale of this neighbourhood will impact the approaches that we might consider for composing operators.

Suppose we wish to compose some point operator $P$ with the discrete Fourier transform $F$; and then apply the resulting pipeline to an input image $f$ such that $F_p = F \circ P \circ f$. Group memory is typically a scarce resource, and so it is unlikely that a naive tiling implementation can be scaled to encompass an entire image. Similarly, rematerialization would require $N^2$ evaluations of $P$, where $N$ is the number of pixels in the input image $f$. Therefore, for large images simply evaluating $P$ and $F$ as separate kernel dispatches is likely to prove the most efficient of the three approaches.

## 2.9 FUNDAMENTALS OF RAY TRACING

Chapter 6 describes RTKit, our new framework for exploring the performance optimization of ray tracing on heterogeneous systems. This work builds upon our C++

compiler and programming model for HSA, described in Chapter 5. As with the preceding section on fundamentals of image processing, this section aims to give a high-level introduction to ray tracing algorithms, and to relate the characteristics of those algorithms to the properties of heterogeneous systems.

Ray tracing algorithms generate images by attempting to simulate light transport between a virtual light source and the image plane of a virtual camera. This is accomplished by forming paths constructed of ray segments between a point on the camera's image plane and some virtual light source, with intermediate vertices representing surface interactions. A variety of methods have been proposed for generating candidate paths.

Early work by Appel (1968) traces rays from the camera into the scene to find the closest visible surface point per pixel in the output image, and then generates secondary ray segments between each visible surface point and each light source. These secondary ray segments are tested against the scene geometry to identify which lights are occluded and generate shadows. Whitted (1979) extends Appel's approach through the recursive use of additional secondary rays to support refraction and reflection. Because these secondary rays are generated as perfect specular reflections and refractions, Whitted ray tracing is limited to sharp appearances. Cook, Porter and Carpenter (1984) describe use of additional rays to simulate fuzzy effects such as depth of field, motion blur and soft shadows. Kajiya (1986) provides further formalization of this approach, describing the rendering equation and introducing Monte-Carlo path tracing to resolve global illumination.

Regardless of the specific algorithm, all ray tracing algorithms share a set of common tasks: ray generation, ray-geometry intersection tests, shading and accumulating samples into a final image.

Rays originating from a camera, or from a point or cone light source, are referred to as primary rays. Rays originating at surface intersection points are referred to as secondary rays.

In aiming to understand the performance implications of extending ray tracing to heterogeneous systems and parallel processors such as GPUs, we must address the properties of sets of rays, rather than individual rays.

A set of primary rays originating from the same camera or light source typically exhibits a high degree of similarity between rays. For example, rays originating from a camera will have near identical points of origin, and their directions will be constrained to the view frustum of the virtual camera. In many cases, the generation

of such a set of primary rays can be further controlled to increase the degree of similarity between neighbouring rays within a set. One approach to this would be to generate rays by sampling the image plane of a camera using a space-filling curve such as a Z-order or Hilbert curve (Sagan, 2012). These curves provide a mapping from a one-dimensional linear sequence of indices to points on the two-dimensional image plane with a higher degree of spatial locality than a simple row-by-row linear mapping. By improving the degree of locality of sample points, the directional locality of the generated rays is also improved at low computational overhead. We can refer to a collection of rays that exhibit a high degree of similarity as *coherent* rays.

By contrast, secondary rays tend to exhibit a lower degree of similarity, or coherence. The points of origin of secondary rays are likely to be more widely distributed spatially, especially for longer paths containing multiple surface interactions. For many techniques, such as distribution ray tracing (Cook, Porter and Carpenter, 1984) or Monte-Carlo path tracing (Kajiya, 1986), the directional component of secondary rays is generated by sampling from a hemisphere oriented around the normal vector of the originating surface. To aid convergence in Monte-Carlo integrators, it is desirable to distribute samples relative to their likely contribution to the final image. Importance sampling techniques like Multiple Importance Sampling (Veach, 1997) are commonly used here. In the case of highly diffuse, or matte, surfaces this distribution is approximately equivalent to a uniform distribution across the visible hemisphere. Consequently, the degree of similarity between the directional components of rays reflected from matte surfaces can be extremely low in Monte-Carlo renderers.

Regardless of the method used to generate candidate rays, all of these techniques rely upon testing ray-segments for intersection with the geometric representation of a three-dimensional scene. Such scenes are typically represented as collections of simpler geometric primitives such as triangles or parametric surface patches. Brute-force searches testing each ray-segment against every geometric primitive within a three-dimensional scene rapidly become intractable as image and scene resolutions scale.

As a result, a number of spatial subdivision techniques have been developed which are able to dramatically reduce the number of ray-primitive tests required in order to find intersection points. These include uniform grids, tree-based structures such as octrees (Glassner, 1984) and kd-trees (Bentley, 1975), and directional techniques (Mortensen et al., 2007).

Bounding Volume Hierarchy (BVH) trees (Kay and Kajiya, 1986; Rubin and Whitted, 1980) are a widely used spatial acceleration structure in GPU ray tracing, and are used as the acceleration structure of choice within RTKit. These data structures are based on trees of nested Axis-Aligned Bounding Boxes (AABBs), and facilitate the rapid identification of ray-geometry intersections through the use of depth-first tree searches. BVH trees handle the non-uniform spatial distribution of geometric primitives better than uniform grids; the use of AABBs makes testing rays for intersection with internal nodes fast; and avoid the build time complexity and numerical stability challenges of techniques such as Binary Space Partitioning (BSP) trees.

We can now relate the behaviour of ray-geometry intersection algorithms to hardware in a heterogeneous system.

Ray tracing is an embarrassingly parallel problem. Multiple paths can be constructed, and their contribution to a final image evaluated, wholly independently of each other. This suggests that it should be trivial to generate and evaluate a large population of paths in parallel.

In practice, whilst paths are independent, the grouping of paths into GPU wavefronts or SIMD vectors leads to two major issues. Firstly, due to the low degree of coherence between secondary rays, traversal of tree-based spatial acceleration structures such as BVH or kd-trees is likely to result in a dissimilar order of node traversal for rays within the same wavefront. This corresponds to poor cache utilization on CPU architectures; along with increased branch divergence and severely reduced memory coalescing on GPU architectures.

By contrast, primary rays can generally be expected to perform well on both CPU and GPU architectures. The high degree of coherence between neighbouring rays in a batch is likely to result in a correspondingly high degree of similarity in the traversal order of nodes within the acceleration tree. This leads to an improved probability that the required nodes will be located within a CPU cache, and that memory accesses can be coalesced on GPUs.

A second issue is that of low SIMD utilization. BVH tree traversal may exit early during ray-geometry intersection testing; and paths may terminate unevenly either due to failure to intersect with geometry, or due to forced termination through Russian roulette in Monte-Carlo path tracers. This problem is particularly severe in wide-SIMD devices such as GPUs, where a single active path may keep a 32 or 64 element wide wavefront live.

When applying a heterogeneous system to ray tracing, we must map the tasks that make up a ray tracing algorithm: ray generation, ray-geometry intersection tests, shading and accumulating samples into a final image; to specific PUs. We may also wish to experiment with varying acceleration structures and traversal techniques, transform data structures to a form corresponding to the optimal SIMD-width for a particular PU or experiment with compaction or sorting to attempt to combat divergence. Determining whether a particular candidate mapping, or potential optimization, is beneficial on a particular heterogeneous system requires experimentation. In chapter 6, we describe RTKit, a framework for exploring such optimizations on heterogeneous systems.

## 2.10 DISCUSSION

In this chapter, we have provided a brief introduction to heterogeneous systems, and to the execution, memory and compilation models commonly found in language runtime implementations for heterogeneous systems.

We have also described two recent specifications for heterogeneous computing: SYCL and HSA.

The SYCL standard provides a single-source C++11-based programming model as a software layer on OpenCL hardware. We have provided a brief overview of the SYCL programming model.

HSA is a lower-level specification, defining hardware requirements, a virtual instruction set and a small runtime API. We have provided an introduction to HSA.

Additionally, we have explored some fundamental concepts from two fields: image processing and ray tracing. In particular we have explored how these concepts can be mapped to the memory and execution models of heterogeneous architectures and the performance implications of these mappings.

This chapter has aimed to provide a base foundation for understanding of heterogeneous systems; an introduction to the standards that we will build upon; and a high-level overview of image processing and ray tracing, which will provide use cases for our work.

In the next chapter, we will provide a more detailed review of the software ecosystem for heterogeneous systems. We will also survey the state of the art for both image processing and ray tracing on heterogeneous systems.

# 3 RELATED WORK

## 3.1 INTRODUCTION

This thesis is primarily concerned with three new works: a Domain Specific Language (DSL) for image processing which builds upon the SYCL standard for C++ on heterogeneous systems; a compiler and programming model for C++ which builds upon the Heterogeneous System Architecture (HSA) standards; and an exploration of accelerating ray tracing on a heterogeneous system using our C++-based programming model. In this chapter we will discuss existing work in each of these fields.

We will begin this chapter by providing an overview of programming models for heterogeneous systems in section 3.2. Programming models and language bindings for heterogeneous systems have been developed for a wide range of languages. In order to limit the scope of this section, and because of our own work focuses on C++, we will primarily focus our discussion on C and C++-based models.

Chapter 5 will describe a C++-based programming model that builds upon HSA. The HSA Foundation envision the HSA runtime library and virtual instruction set as providing a foundation to enable the development of parallel programming models on hardware which supports HSA. Our work on developing a C++ programming model for HSA is one of the first such examples. Concurrent with our work, other authors have also produced works utilizing HSA to provide acceleration. We will review these works in section 3.3.

We will follow this in section 3.4 with a discussion of DSLs for image processing on heterogeneous systems. Efficient programming of heterogeneous systems remains a complex task, requiring significant machine knowledge. This can make it challenging for experts in other domains, such as image processing or machine vision, to achieve maximum efficiency for their algorithms. DSLs provide one possible solution to this problem. They enable us to provide a high-level abstraction which provides familiar syntax and nomenclature to domain experts, whilst allowing machine experts to

optimize the low-level behaviour of the generated code. Chapter 4 will describe our image processing DSL, which builds upon SYCL.

One potential source of performance issues when attempting to produce libraries of Graphics Processing Unit (GPU) kernels for use by domain experts is memory traffic caused by a combination of GPU kernel execution models and the use of fine-grained kernels. These kernel execution models require that the results of computation within a kernel are written to GPU global memory. Subsequent kernels consuming these values must then reload their inputs from global memory. This memory is typically off-chip and carries a high cost of access, both in terms of power and latency (Dally, 2011).

Kernel fusion is one solution to this issue. By combining multiple kernels into a single kernel, the results of intermediate computations can be retained in registers. This reduces power and memory bandwidth requirements and often leads to improved performance. Whilst this fusion can be achieved through manual effort, it requires machine expertise. This problem is further exacerbated in the case of kernel libraries, where the specific combination of kernels that a third-party programmer wishes to use may not be known a priori.

A major motivation for our work on DSLs was to explore whether SYCL's programming model was sufficient to enable kernel fusion without the need for additional offline tools. As such, we follow our discussion of DSLs with a review of kernel fusion techniques in section 3.5.

Ray and path tracing (Kajiya, 1986; Whitted, 1979) form a class of computer graphics techniques that are widely used for realistic rendering. Ray tracing serves as an exemplar use-case for our HSA-based programming model. It both provides a larger application on which to evaluate our programming model, and serves to illustrate the additional complexity of targeting heterogeneous systems. These algorithms are highly parallel, computationally intensive and lead to memory access patterns that prove challenging for GPUs. As an important and computationally demanding task, a significant body of work has been dedicated to approaches to accelerating ray tracing algorithms. Despite this, little work has been conducted exploring the suitability of shared memory heterogeneous systems such as Advanced Micro Devices (AMD)'s Accelerated Processing Units (APUs) for accelerating ray tracing. In chapter 6, we will describe the implementation of a new ray tracing framework for heterogeneous systems. This framework relies upon the flexibility and ease code reuse provided by our C++ programming model for HSA to aid the exploration of the optimization

space. In section 3.6, we will explore existing approaches to exploiting heterogeneous systems to accelerate ray tracing.

Finally, we conclude the chapter with a few final remarks in section 3.7.

## 3.2 SOFTWARE ECOSYSTEM FOR HETEROGENEOUS SYSTEMS

We will begin with a review of major runtimes and programming models for heterogeneous systems, with a focus on C and C++-based models. Graphics processors and other accelerators have advanced rapidly in terms of both performance and capabilities over the last decade. Over the same period programming models for targeting such devices have also evolved considerably.

Early examples such as Brook for GPUs (Buck et al., 2004) made use of graphics shading languages such as Cg (Mark et al., 2003) as an intermediate representation. Sh (M.D. McCool, Qin and Popa, 2002; M. McCool et al., 2004) provides one of the earliest examples attempting to bring C++ to GPUs. With Sh, M. McCool et al. demonstrated an early example of embedding a shading language within host C++ code. This was later commercialized as RapidMind (Monteyne, 2008).

These languages were followed by the release of early proprietary toolkits such as CUDA (NVIDIA Corporation, 2007), and ATI Technologies Incorporated (ATI) Stream (Advanced Micro Devices, 2008) in 2007. Both CUDA and Stream were proprietary Application Programming Interfaces (APIs), each enabling the offloading of computation to discrete GPUs produced by a single hardware vendor.

Proprietary solutions continue to hold significant market share, particularly CUDA. More recently, we have also seen a rise in standardization efforts by industry consortiums such as the Khronos Group with OpenCL (Khronos OpenCL Working Group, 2008); the HSA Foundation with HSA (HSA Foundation, 2015a,b,c) and the OpenMP Architecture Review Board (ARB) with OpenMP 4 (OpenMP ARB, 2015).

In comparing the current ecosystem of heterogeneous programming models and runtimes, we observe some common features that appear in multiple models and runtimes. We can make use of these features to broadly classify models into three tiers of abstraction: low, mid and high-level.

### 3.2.1 Low-Level APIs

At the lowest level of abstraction, we find the APIs which give the greatest level of programmer control. These APIs typically feature a C-based runtime API and separate kernel or shader languages.

These APIs commonly expose functionality for enumerating accelerator devices in a system; memory allocation and data movement; kernel compilation; work dispatch; and synchronization primitives.

These APIs typically utilize dual-source models and specialized kernel languages. These may be problem domain-focused high-level languages such as OpenCL C (Khronos OpenCL Working Group, 2016), HLSL (Oneppo, 2007) or GLSL (Khronos OpenGL Working Group, 2016b); alternatively they may be intermediate languages such SPIR (Khronos OpenCL Working Group – SPIR subgroup, 2014), SPIR-V (Khronos SPIR-V Working Group, 2016) and Heterogeneous System Architecture Intermediate Language (HSAIL) (HSA Foundation, 2016b).

Due to tight integration with underlying device drivers, these models are either proprietary solutions designed by hardware or operating system vendors, or the product of industrial consortiums such as the Khronos Group or HSA Foundation.

CUDA represents the most dominant General Purpose Graphics Processing Unit (GPGPU) framework. CUDA is proprietary toolkit, with a closed-source compiler and targeting hardware from a single vendor. The initial implementation of CUDA supported only the C subset of C++, but parsed according to C++ syntax rules (NVIDIA Corporation, 2007, p. 22). Subsequent versions of CUDA have progressively increased the level of C++ support, with CUDA 8 supporting C++11 with some constraints. CUDA supports two API models; a low-level driver API that matches many of the characteristics of a low-level model, and a runtime API, which might more accurately be categorized as a mid-level model. CUDA supports single-source compilation targeting a host processor and NVIDIA GPUs. More recently, GPUCC (Wu et al., 2016) has provided an open-source compiler for CUDA, based on LLVM (Lattner and Adve, 2004) and Clang (Lattner et al., 2007). However, GPUCC still only targets NVIDIA GPUs. We provide a more detailed comparison of CUDA and our own C++ programming model for HSA in chapter 5.

The Khronos Group released OpenCL (Khronos OpenCL Working Group, 2008), an open and cross-vendor standard for GPGPU programming in 2008. Unlike CUDA, OpenCL support is available for a wide range of hardware devices including Central

Processing Units (CPUs) (*Intel® SDK for OpenCL™ Applications*), GPUs (*APP SDK – A Complete Development Platform - AMD*), Field-Programmable Gate Arrays (FPGAs) (*Intel® FPGA SDK for Open Computing Language*) and Digital Signal Processors (DSPs) (*TI OpenCL v01.01.xx - TI OpenCL Documentation*). OpenCL originally defined a C99-based kernel language, OpenCL C (Khronos OpenCL Working Group, 2008). This was later augmented with the addition of an intermediate language, Standard Portable Intermediate Representation (SPIR) (Khronos OpenCL Working Group – SPIR subgroup, 2014). This enables compiler developers to target a bitcode representation derived from that found in the LLVM (Lattner and Adve, 2004) toolchain.

Whilst OpenCL initially relied upon OpenCL C for a kernel language, several different projects have subsequently addressed C++ support for OpenCL. In "OpenCL C++", Gaster and Howes (2013) describe an approach to encapsulating the OpenCL API in modern C++, along with a C++11-based kernel language. They also provide the first example of extending OpenCL's address spaces to C++11. Looking forwards to the possibility of future devices with unified Shared Virtual Memory (SVM), Gaster and Howes recognized the importance of a common pointer type which could be shared between host and device code. They define a custom pointer class to resolve this. Although this unified pointer type is provided, it is separate from native pointers and thus does not allow interoperability with existing libraries without modification. This approach is slightly more intrusive than our remapping of the generic address space, discussed in section 5.4.2.

More recently, the Khronos Group have released a further specification enabling the use of C++ on OpenCL accelerators as part of OpenCL 2.2. Also titled OpenCL C++ (Khronos OpenCL Working Group, 2017), but separate from the work of Gaster and Howes, this provides a kernel-only language using a static subset of C++. As a kernel-only language, this approach continues the dual-source approach adopted with OpenCL C. Although this provides many of the benefits of C++, such as classes and templates, it still requires separate host and device languages. One significant restriction of the OpenCL C++ kernel language relates to overloaded or templated functions. In this context, multiple functions share a common identifier in the source code, but unique type signatures. C++ compilers are able to generate unique symbol names for each function by augmenting the original identifier with additional annotations based on the type signature, through a process referred to as name mangling. Unfortunately, these mangled names are not easily human-readable. In the interests of ease of interoperability with the OpenCL host API, the Khronos OpenCL Working Group elected to prohibit the use of name mangling on kernel functions, and

consequently kernel functions cannot be templated or overloaded (Khronos OpenCL Working Group, 2017, p. 43).This makes integration into libraries which make heavy use of template metaprogramming, such as Eigen (Guennebaud, Jacob et al., 2010) or the DSL we describe in chapter 4 challenging.

We can also place HSA's runtime API (HSA Foundation, 2015c) in this tier. Of the APIs discussed in this section, HSA provides the smallest feature set and the lowest level of abstraction. We can attribute this to philosophical differences in approach. HSA aims to provide a low-level toolset focused on the needs of developers of parallel and heterogeneous programming models, rather than application developers. Consequently, it lacks features such as a high-level kernel language, a library of built-in mathematical functions or simple APIs for kernel dispatch.

An analogous example to HSA can be found in the field of dedicated graphics API's. Vulkan (Khronos Vulkan Working Group, 2016) provides a similar low-level toolset to HSA, exposing hardware related concepts such as GPU dispatch queues, synchronization primitives and memory regions to application developers. Like HSA, shader input is accomplished via a bytecode-based intermediate language - SPIR-V (Khronos SPIR-V Working Group, 2016), rather than a high-level human-readable language like OpenGL Shading Language (GLSL) (Khronos OpenGL Working Group, 2016b). This approach lowers the complexity of implementation and performance overheads of the API implementations, when compared to alternative graphics API's that have adopted a higher level of abstraction, such as OpenGL (Khronos OpenGL Working Group, 2016a). However, this comes at the cost of increased complexity for application developers.

We can also view the compute capabilities of both DirectCompute (Ni, 2009) and Metal (Apple, 2014) as further examples of these low-level APIs.

Due largely to the ubiquity of C as a system programming language, the APIs described above all primarily treat C, and consequently C++, as their primary target language. Despite this, a wide range of projects exist to provide relatively thin wrappers and binding interfaces to alternative host languages. Examples of this include PyOpenCL and PyCUDA (Klöckner et al., 2012), providing Python bindings for OpenCL and CUDA; the Erlang (Rogvall, 2009) and Ruby (Videau, 2013) OpenCL language bindings projects; and the Vulkano (Krieger, 2016) project, providing Vulkan bindings for Rust.

### 3.2.2 Mid-Level APIs

Above the low-level APIs, we can group several C++-based programming models which are still strongly tied to the underlying implementations. These models are typically implemented by building upon the low-level APIs described in the preceding section. Concepts from the low-level APIs are still exposed to developers, often through some form of abstraction.

Whilst these models introduce abstractions, they do not free the application developer from the need to be aware of common concepts in programming heterogeneous devices. Concepts such as accelerator devices, dispatch queues and device-local memory buffers are still present.

We can refer to OpenCL and SYCL's representations of memory buffers for a concrete example of this principle in practice. In SYCL, buffers are strongly typed classes with familiar array syntax; and which perform memory management and data transfers implicitly. In OpenCL, a buffer is an untyped block of memory which must be explicitly allocated; copied to and from the accelerator device; and destroyed by the application developer. In this sense, SYCL provides some level of abstraction over OpenCL, but fails to make the properties of the underlying platform disappear completely.

At this level, we see single-source models begin to appear. In these models, the early stages of kernel compilation such as the parsing and semantic analysis of high-level source languages are predominantly treated as an offline process. This is then combined with runtime finalization from some intermediate format to native accelerator Instruction Set Architectures (ISAs), with this transformation being performed transparently by the language runtime library. Implicit management of data movement by the language runtime is also common in this tier.

Many of these models are library or language-based, introducing specialized container types to represent memory buffers accessible to accelerator devices, and in some cases additional language keywords to annotate host and device code. Listing 3-1 provides an illustrative example of this, written in C++ AMP (Microsoft Corporation, 2013). However, many of the other examples discussed below bear strong syntactic similarities.

SYCL (Khronos OpenCL Working Group – SYCL subgroup, 2015) is an open standard providing a single-source C++11-based programming model for heterogeneous systems, building upon OpenCL. Unlike CUDA, which by default requires programmers

```
using namespace concurrency;

const size_t N = 1024;
float a[N], b[N], c[N];

array_view<float, 1> av(N, a);
array_view<const float, 1> bv(N, b);
array_view<const float, 1> cv(N, c);
parallel_for_each(extent<1>(N), [=](index<1> idx) restrict(amp) {
  av[idx[0]] = bv[idx[0]] + cv[idx[0]];
});
```

**Listing 3-1:** Example of a Library-based Model - Vector Addition in C++ AMP

to explicitly schedule data movement, SYCL provides abstractions which allow the runtime library to manage data movement both between host memory and accelerator device memory, and between accelerators. There are presently three implementations: ComputeCpp[1] (Codeplay Software Ltd., 2016), triSYCL[2] (Keryell, 2015) and SYCL-GTX[3] (Žužek, 2016). Due to the relatively recent release of the specification, at the time of writing none of these implementations offers a complete, conformant implementation.

Microsoft introduced C++ AMP (Microsoft Corporation, 2013), a single-source C++ implementation targeting DirectCompute. Further implementations have since been demonstrated utilizing OpenCL as a backend (Sharlet et al., 2012). Like SYCL, C++ AMP provides a single-source programming model relying on specialized container classes to enable implicit data movement between host and accelerator memory.

Heterogeneous Compute Compiler (HCC) (Sander et al., 2015) represents the closest work to ours. HCC implements the Heterogeneous Compute (HC) programming model, an extended form of C++ Accelerated Massive Parallelism (C++ AMP)'s model with specific functionality to capitalize on the features of HSA. Like our model, pointers can be transparently shared between host and device code. Some effort has been undertaken to relax the constraints of C++ AMP and remove the need for device function annotations such as `restrict(amp)`, although this work is incomplete. We provide a more detailed comparison of SYCL, HCC, CUDA and our model in chapter 5.

1 https://www.codeplay.com/products/computesuite/computecpp
2 https://github.com/Xilinx/triSYCL
3 https://github.com/ProGTX/sycl-gtx

Æcute (Howes et al., 2009a,b) provides an approach to composing descriptors describing memory access patterns; iteration spaces describing execution schedules; and kernels describing a computation applied over the iteration space. Data movement schedules can then be derived by the compiler and runtime from the descriptors. Howes et al. (2009a) initially applied this approach to scheduling Direct Memory Access (DMA) transfers for the Cell Broadband Engine (BE) and later extended it to CUDA GPUs (2009b). The system of buffers and accessors later adopted by SYCL for deriving memory transfer schedules bears strong similarities to this approach.

PACXX (Haidl and Gorlatch, 2014) is a C++14-based single-source programming model, targeting OpenCL and NVIDIA GPUs. The PACXX compiler embeds an extended form of LLVM IR into the compiled executable. PACXX has independently made many similar design decisions to our model, which we feel validates both works. Unlike our compiler, PACXX resolves address spaces at the code generation stage as transformation passes applied to the LLVM IR. Whilst this approach has the attractive property of allowing alternative frontend languages to be implemented with relative ease in the future, it also prohibits the implementation of features such template specialization on address spaces which is possible in our model. HLSF (Dütsch et al., 2014) builds upon PACXX, providing a C++ framework for generating stencil code kernels from high-level C++ constructs. Stencil codes are a class of iterative kernel, where each element of an output array is computed by sampling some neighbourhood of an input array according to a fixed pattern. These codes have many applications, notably including the local image processing operators described in section 2.8. In this sense, HLSF shares some of the same challenges as our image processing DSL, described in chapter 4. However, this work only discusses the generation of single stencils, and does not address composition of more complex expressions.

Like SYCL and C++ AMP, PACXX builds upon lower-level APIs which feature explicit memory copies; OpenCL and CUDA. Consequently, the PACXX runtime is forced to perform memory transfers between the host processor and the accelerator devices. PACXX aims to make these transfers implicit and managed by the runtime. However, unlike SYCL and C++ AMP, PACXX accomplishes this by providing customized implementations of `std::vector` and `std::array` class, rather than introducing new container types. This differs from the approach supported by our model, and that of HC. Due to the ubiquity of shared virtual memory on HSA, we can operate directly on data without copies.

Beyond the realm of C++ programming models, similar approaches have been adopted to provide parallel programming models for other languages. JCUDA (Yan, Grossman and Sarkar, 2009) aims to bring CUDA's higher-level runtime API programming model to Java. Aparapi (Syncleus, 2016) represents an alternative approach to executing Java code on GPUs, albeit with a design that is more familiar to Java developers, and less closely modelled on emulating CUDA. In a similar spirit, DCompute (Wilson, 2017) is an effort to define a single-source programming model for D.

Our C++-based programming model for HSA also fits into this category of mid-level models. It builds upon the HSA runtime to provide a single-source programming model, whilst still exposing the underlying concepts of the platform such as queues and synchronization objects. The primary area in which our model diverges from models such as C++ AMP and SYCL relates to the representation of memory buffers and the use of specialized container types. Due to the pervasive use of SVM in HSA, we can dispense with these types entirely, and simply use standard pointers to share data between agents. Additionally, our use of HSA allows us to relax some on the restrictions on concurrent access to data from multiple agents found in other programming models.

Further examples of these mid-tier models include CUDA's Runtime API which builds upon CUDA's Driver API; and Boost.Compute (*Boost.Compute*) which abstracts OpenCL.

So far the solutions that we have discussed have all been language and library-based. Directive-based models provide an alternative approach. In these models, loops within a sequential C++ program can be augmented with additional metadata in the form of compiler pragmas, instructing the compiler to parallelise the annotated loops.

Listing 3-2 provides an illustrative example of this approach, using a vector addition implemented in OpenMP 4. In contrast to the library-based C++ AMP example shown in listing 3-1, no specialized container types are required, and the conventional representation of loops are retained in the source code, rather than being abstracted into an iteration function. In this example, pragmas indicate to the compiler that two of the arrays should be mapped to the accelerator device before the loop is evaluated; that the loop should be evaluated in parallel on an accelerator; and that the result array should be copied back to host memory after evaluation of the loop is completed.

```
const size_t N = 1024;
float a[N], b[N], c[N];

// Copy arrays b, c to accelerator, copy array a back on completion.
#pragma omp target data map(to: b[0:N], c[0:N]) map(from: a[0:N])
{
  // Execute the loop in parallel on the accelerator device.
  #pragma omp target
  #pragma omp parallel for
  for (auto i = 0; i < N; ++i)
    a[i] = b[i] + c[i];
}
```

**Listing 3-2:** Example of a Directive-based Model - Vector Addition in OpenMP 4

Other examples of directive-based approaches include OpenMPC (S. Lee and Eigenmann, 2010), OpenMP (OpenMP ARB, 2015), OpenACC (OpenACC Working Group and others, 2011), PGI Accelerator (Wolfe, 2010), hiCUDA (Han and Abdelrahman, 2009, 2011), HMPP (Dolbeau, Bihan and Bodin, 2007) and Intel MIC (Duran and Klemm, 2012; Newburn et al., 2013).

Support for this directive-based approach is not limited to C and C++. A number of authors and compiler vendors have applied this approach to Fortan (Sung et al., 2017; Wallcraft, 2002). The application of directive-based techniques to both C and Fortran is particular popular in the field of scientific computing, where it provides a relative inobtrusive method of accelerating code originally authored by domain scientists.

### 3.2.3 High-Level Abstractions

At the highest level of abstraction, we have works which seek to insulate the programmer from low-level architectural details and provide portability between systems. These approaches differ from our own work, in that they aim to abstract the underlying machine and so provide portability. However, they could reasonably be implemented as an abstraction layer above our HSA-based programming model.

NVIDIA's Thrust (Bell and Hoberock, 2011) provides abstractions over host and device-side vectors, and a set of functions for common parallel patterns such as transforms, scans and reductions. Thrust builds upon a number of different underlying APIs and runtimes to provide backends for computation. CUDA (NVIDIA Corpor-

ation, 2007) acts as the default backend, but OpenMP (OpenMP ARB, 2015), Intel Threading Building Blocks (TBB) (Intel, 2016), and standard C++ runtimes are also supported. This illustrates the key value of these high-level abstractions. Application developers are able to target a single interface, but can port code to a variety of architectures with minimal changes.

Parallel Standard Template Library (STL) (ISO/IEC, 2015), a proposed extension to C++17, extends many existing algorithms from the C++ standard library with an execution policy argument. This policy argument allows C++ programmers to indicate whether they wish these algorithms to execute sequentially, in parallel, or in a interleaved parallel form suitable for vectorization. The interfaces to these algorithms use standard C++ iterators rather than specialized containers such as SYCL's `buffer` or C++ AMP's `array_view`. Unlike the runtimes in the preceding section, the use of an accelerator device is completely transparent to the application programmer. The C++ standard library implementation is responsible for handling the complexity of accelerator device selection, data movement and the launching and synchronization threads of execution. Parallel STL is strongly influenced by a variety of vendor-specific C++ implementation efforts such as Intel's TBB (Intel, 2016), Microsoft's Parallel Patterns Library (PPL) (Campbell and Miller, 2011) and C++AMP (Microsoft Corporation, 2013), and NVIDIA's Thrust (Bell and Hoberock, 2011).

In chapter 5, we provide comparative benchmarks for Parallel STL running as an abstraction above SYCL, HCC and our C++ programming model for HSA. This serves as a further illustration of how these high-level models can provide portability by abstracting the underlying lower-level runtimes.

In the field of High Performance Computing (HPC), projects such as Kokkos (Edwards and Trott, 2013), RAJA (Hornung, Keasler et al., 2014) and High Performance ParalleX (HPX) (Kaiser, Adelstein-Lelbach et al., 2016; Kaiser, Heller et al., 2014) aim to define C++-based parallel programming models which abstract away the details of the underlying machine.

Kokkos (Edwards and Trott, 2013) provides a model based on parallel patterns such as for-each, scans and reductions; multi-dimensional array container types; and the parallel execution of C++ function objects which perform manipulation of those arrays. Through the use of C++ templates, the internal data layout of the arrays can be configured to suit the characteristics of the specific algorithm and underlying hardware. Kokkos is expressed in standard C++, and transparently provides

OpenMP (OpenMP ARB, 2015), POSIX Threads and CUDA (NVIDIA Corporation, 2007) backends.

Similar to Kokkos and Thrust, RAJA (Hornung, Keasler et al., 2014) provides a set of parallel patterns, combined with execution policies and index sets to control the mapping of loop iterations to underlying hardware. The core programming model is similar to that expressed in many of the previously discussed models, including Kokkos, SYCL and C++ AMP. Lambda functions are used to express loop bodies. Index sets then enable the partitioning of the iteration space. For example, a mapping for multi-core CPU might map large dense regions of the iteration space to specific CPU cores, while a mapping for GPUs might map each element in the iteration space to a unique work-item.

Looking beyond C++, a number of libraries and languages aim to provide transparent GPU acceleration without requiring significant user intervention. This is especially true in the field of array languages. Theano (Bastien et al., 2012; Bergstra et al., 2010) transparently generates and executes CUDA kernels from mathematical expressions over multi-dimensional arrays. In a similar vein, Accelerate (Chakravarty et al., 2011) is an embedded array language, hosted in Haskell, and providing support for both multi-core CPUs and GPUs. We can also look to deep-learning frameworks as a strong domain-specific example, with frameworks such as TensorFlow (Abadi et al., 2016), Caffe (Jia et al., 2014) and Torch (Collobert, Kavukcuoglu and Farabet, 2011) all providing easy access to GPU acceleration without requiring machine knowledge from data scientists.

## 3.3 LANGUAGES TARGETING HETEROGENEOUS SYSTEM ARCHITECTURE

Chapter 5 describes our C++-based programming model for HSA. Due in part to the relatively recent release of specifications and limited availability of HSA runtime implementations, little work has been demonstrated on languages which target HSA. Concurrent with our own work, a number of other authors have developed compilers and runtimes with support for HSA.

The majority of the languages described below utilize derivatives of the same LLVM backend as is found in our compiler to perform HSAIL code generation [4]. However, our backend has undergone extensive independent development.

Aside from our own work, CL Offline Compiler (CLOC) (Rodgers, 2015) represents one of the earliest examples of a compiler for HSA. CLOC does not attempt to provide a complete and cohesive programming model. Instead CLOC provides a simple tool to generate HSAIL code via offline compilation of OpenCL C. A companion project, SNACK provides a simple API for calling the generated HSAIL kernels from C. CLOC and SNACK provide a much more limited programming model than our model, but does provide a simple route to integrating via OpenCL C kernels into an application.

Whilst CLOC is able to compile OpenCL C kernels, it is not a full OpenCL runtime. POCL (Jääskeläinen et al., 2014) is an open-source OpenCL runtime implementation with experimental support for HSA. Yang et al. (2015) explored OpenCL performance on HSA by porting the POCL to target HSA. More recently, AMD have released an experimental OpenCL 1.2 runtime as part of their ROCm platform (Advanced Micro Devices, 2016c).

HC is a C++14-based programming model which extends C++ AMP with functionality to support HSA. By aiming to providing a single-source C++ compiler and programming model for HSA, HC is the closest work to our own work, and has made many similar design decisions. A detailed comparison of HC and our programming model can be found in chapter 5. GNU Compiler Collection (GCC) 6 also introduces partial support for utilizing HSA to accelerate OpenMP (Jambor, 2015).

HSA is not limited to just natively compiled languages, the Java-based Aparapi project is a parallelism API that can run lambda functions on an HSA device (*Adding HSA Support to Aparapi lambda branch*). Similarly, Numba (Lam, Pitrou and Seibert, 2015) is an LLVM-based compiler for Python. Numba is able to use Just In Time (JIT) compilation to generate HSA kernels from annotated functions.

4 https://github.com/HSAFoundation/HLC-HSAIL-Development-LLVM

## 3.4 DOMAIN SPECIFIC LANGUAGES FOR IMAGE PROCESSING ON HETEROGENEOUS SYSTEMS

Chapter 4 describes a DSL that utilizes SYCL to transparently accelerate image processing on systems supporting OpenCL.

Many use cases for image processing and machine vision impose challenging performance constraints. Portable devices such as smartphones and tablets are power constrained and require efficient implementations to preserve battery life and avoid thermal throttling. Industrial safety and Advanced Driver Assistance Systems (ADAS) systems have hard timing constraints, as do applications in the gaming and entertainment domains such as optical motion capture and augmented reality. Medical imaging often deals with huge volumes of high resolution data.

Parallel and heterogeneous computing appears to offer a potential solution to these challenges. However, application development for heterogeneous systems is itself a challenging and complex field. The major programming models for heterogeneous computing, such as CUDA and OpenCL, tend to assume familiarity with the underlying hardware architecture. As we discussed in section 3.2.2, more abstract models such as SYCL and C++ AMP reduce the level of machine knowledge required from application developers but do not entirely eliminate it.

Libraries of domain-specific primitives, such as OpenCV (Itseez, 2015), NVIDIA Performance Primitives (NPP) (NVIDIA Corporation, 2011), OpenCLIPP (Akhloufi and Campagna, 2014) and Intel Integrated Performance Primitives (IPP) (Taylor, 2007) offer potential solutions to some of these problems. These libraries offer collections of functions, optimized by machine experts. Whilst machine experts can optimize individual elementary functions within such libraries, optimizing the macro scale based on how the elementary functions will be combined is challenging. This is due to a lack of prior knowledge of how application developers will choose to compose functions.

Distinct from libraries of domain-specific primitives, DSLs provide a possible approach to resolving this problem. DSLs are programming languages focused on the requirements of a specific application domain, such as image processing. This is distinct from general purpose languages, such as C or Python, which aim to be applicable to a wide range of problem domains. The use of DSLs can enable domain experts to work in high-level languages which express concepts and syntax specific to the application field, whilst machine experts can apply knowledge of the underlying

hardware architecture to generate efficient implementations. By using a language-based approach, as opposed to an API-based approach, a toolchain gains further information regarding the manner in which a third-party application developer has chosen to combine elementary operations, and consequently has further opportunities for optimization.

Domain-specific languages have been applied in this manner to a wide range of problem domains including image processing, scientific visualization (Chiw et al., 2012; Choi et al., 2014; Kindlmann et al., 2016), machine learning (Sujeeth et al., 2011) and physical simulation (Bernstein et al., 2016; Kjolstad et al., 2016).

DSLs can be categorized into two classes, based on implementation approach and usage. External or free-standing DSLs are domain-specific languages with their own dedicated compiler or interpreter tooling. This provides for great flexibility, allowing such languages freedom to define their own syntax and parsing rules. This comes at the cost of requiring the language authors to implement of a more significant proportion of the necessary compilation machinery. By contrast, internal or embedded DSLs are hosted within a general-purpose language. In this case, the syntax of the DSL must be expressed in terms of valid constructs within in the hosting language. This reduces the implementation burden when developing the DSL itself, as parsing and code-generation can now be handled by the existing compilation machinery of the hosting language.

We can further subdivide embedded DSLs based on the depth of the embedding within the host language. *Deep embedding* involves an approach whereby expressions within the DSL are used to construct an abstract syntax tree or similar structure, which can be subsequently manipulated for optimization purposes, or traversed in order to evaluate the expressions. This approach mirrors the traditional frontend phases of a compiler, but with parsing handled by the existing hosting language tooling. Semantic analysis, additional optimizations and expression evaluation are then implemented by the language authors in terms of transformations applied to the abstract syntax tree. Under this approach, terms within the DSL map to syntactic elements. A *shallow embedding* dispenses with the use of an abstract tree or similar structure. Instead terms within the embedded DSL are implemented directly in terms of their semantic behaviour. Shallow embeddings allow for easier maintenance and extension, as they don't require the implementation and modification of an Abstract Syntax Tree (AST). This comes at the cost of reduced flexibility with regard to evaluation, with deep embeddings being more amenable to lazy evaluation or performing additional optimizations through manipulation of the generated AST.

Our DSL, described in chapter 4, is a deeply embedded DSL. An embedded approach allows us to avoid the need for additional tools beyond the SYCL compiler. A deep embedding is valuable to our use case for two reasons. Firstly, the construction and manipulation of an AST is the process through which we accomplish fission and fusion of kernels. Secondly, a deeply embedded approach allows us to separate the declaration of expressions from the evaluation. This delayed-evaluation allows us to express an interface entirely in terms of standard host-side C++ and still make use of device-side kernels and GPU execution at a later point.

A wide range of different approaches have been applied to the problem of parallel or heterogeneous image processing.

Designing and implementing parallel DSLs requires considerable compilation machinery, and optimization of such DSLs requires specialist machine knowledge. Delite (Brown et al., 2011; Chafi et al., 2010) and AnyDSL (Leißa, Boesche et al., 2015; Leißa, Köster and Hack, 2015) provide frameworks for building parallel DSLs capable of targeting heterogeneous hardware. These two frameworks do not focus specifically on image processing, but rather on the challenge of providing representations that are suitable for the targeting a range of parallel hardware. Both frameworks aim to provide separation between the syntactic expression of DSLs and the mapping to hardware specific constructs by transforming the source DSLs into a domain-agnostic representation before mapping to the target machine.

Halide (Ragan-Kelley et al., 2013) aims to tackle similar problems to Delite and AnyDSL, but with a somewhat different approach. Where Delite and AnyDSL aim to provide tools for separating the machine expertise demands of optimization from the syntactic aspects of DSL design, Halide focuses on a DSL for a single use-case: image-processing, and on the problem of scheduling operations over one-, two- and three-dimensional grids. Halide is a purely functional DSL deeply embedded in C++. It combines a functional representation of an image processing pipeline with operations to manipulate the scheduling of the pipeline to allow rapid iteration when optimizing scheduling. Whilst our approach bears conceptual similarities to that of Halide, with the embedding of a DSL within C++, the implementation approaches differ.

Halide's approach allows for dynamic construction of kernels at runtime, enabling the modification of kernels based on runtime values. This is not possible under our compile-time approach. Conversely, basing our approach on template metaprogramming yields much stronger type-safety than is found in Halide.

Halide's approach allows for greater flexibility with respect to scheduling kernels. However, this comes at the cost of implementing considerable compilation functionality within the language runtime. Under our approach, much of this burden can be placed on the SYCL compiler. The Halide runtime provides backends to target multiple architectures, including CUDA, OpenCL and CPUs. By building upon SYCL, our DSL is able to provide both OpenCL and CPU backends by delegating the work to the underlying SYCL implementation. Further architectures could also reasonably be supported by an extended SYCL implementation, such as Vulkan (Khronos Vulkan Working Group, 2016), HSA or native NVIDIA GPU support via Parallel Thread Execution (PTX) and the low-level CUDA driver API (NVIDIA Corporation, 2007).

Comparative examples between Halide, OpenCV and our DSL can be found in chapter 4.

PolyMage (Mullapudi, Vasista and Bondhugula, 2015) utilizes a DSL embedded within Python to describe multi-stage image processing pipelines. Polyhedral compilation (Bastoul, 2004) and auto-tuning are used to perform optimization across the complete pipeline.

ImageCL (Falch and Elster, 2016) is a standalone DSL which aims to help tackle the problem of performance portability on OpenCL. The input DSL resembles an abstracted form of OpenCL C, with a simplified memory hierarchy and index space. A source-to-source compiler is used to transform the input DSL into multiple candidate OpenCL C kernels with differing sets of optimizations. This is combined with a machine learning based auto-tuner to aid in rapidly selecting efficient kernels for specific architectures. Where both Halide and our DSL (chapter 4) are able to address the fusion of multiple subexpressions into larger GPU kernels, ImageCL focuses on individual kernels rather than complete image processing pipelines. However, one strength of ImageCL's approach over our own is the flexibility to generate multiple specialized kernels for different hardware platforms through the use of auto-tuning. Our use of the C++ template system to express fusion results in a representation that would be difficult to manipulate in this manner. Whilst ImageCL itself deals solely with kernel code generation through the use of source-to-source compilation and not with the scheduling and execution of such kernels, it can be combined with FAST (Smistad, Bozorgi and Lindseth, 2015), a framework for medical visualization which includes support for generating image processing graphs and is able to transparently handle data movement between accelerators and the host processor. Whilst

the primary focus of FAST is visualization, in this context FAST also serves a similar role to SYCL in managing data movement and kernel execution.

HIPA$^{CC}$ (Membarth et al., 2016) aims to generate efficient target specific image processing code from a DSL embedded within C++. The HIPA$^{CC}$ is expressed as C++ classes such that they may be compiled by a standard C++ compiler. This is combined with an augmented compiler, based upon Clang, which is capable of recognising HIPA$^{CC}$ classes. This compiler traverses the generated AST, and matching and transforming HIPA$^{CC}$ into target code for OpenCL, CUDA or RenderScript. This is combined with a target model, allowing the compiler to adjust memory layouts and hierarchy usage, apply optimizations such as loop unrolling or thread coarsening or adjust boundary handling. This approach differs from that adopted by Halide, in that it relies upon an augmented compiler recognising specific AST patterns generated from otherwise standard C++. It also bears similarities to Æcute (Howes et al., 2009a,b) in that access and index-space descriptors expressed as C++ classes are a fundamental component of the DSL. Like ImageCL, HIPA$^{CC}$ focuses on target specific optimization for single kernels, rather than the larger pipelines addressed by Halide, PolyMage and our DSL.

Works by Cornwall et. al.(Cornwall, Beckmann and Kelly, 2006; Cornwall, Howes et al., 2009; Cornwall, Kelly et al., 2007) describes the use of a C++ source-to-source translator to automatically analyse and parallelize an existing image processing library. This project bears some similarities to HIPA$^{CC}$, in that it performs analysis of an AST generated from standard C++ in order to identify regions of code for which it can generate GPU kernels. Also like HIPA$^{CC}$ and Æcute, specialized indexer classes are used to carry information about data dependencies and aid in guiding both the generation of GPU kernels and loop fusion. However where HIPA$^{CC}$ attempts to match and optimize patterns generated from classes defined in it's own embedded DSL, these works aim to identify parallelisable regions located within an existing C++ visual effects library that was not written with GPU acceleration in mind. Whilst this input source code does share some common structure, it also contains patterns that are challenging to parallelise. Consequently, considerable analysis is devoted to identifying and reconstructing parallelisable regions.

KernelGenius (Lepley, Paulin and Flamand, 2013) provides another example of a standalone DSL generating OpenCL kernels for image processing. The syntax of the DSL enables developers to describe nodes within a Directed Acyclic Graph (DAG), expressing both the computation and data dependencies of filters or operators. These nodes can then be combined to form fused OpenCL kernels. KernelGenius does not

generate complete programs. Instead each DAG or kernel generates a well-defined C API, enabling users to set kernel arguments or dispatch an instance of a kernel. Creation of OpenCL buffers and data movement remains the responsibility of the application developer.

Whilst these works adopt differing implementation approaches, we can identify some recurring themes. The conceptual model of an image processing pipeline as a DAG of pure functions is repeated throughout these works, along with the goal of efficiently combining operators in order to improve performance. These themes will also recur in our own work in chapter 4. Similarly, the need to track data dependencies in order to combine operators is a recurring theme, either through the use of additional metadata as described by Cornwall et. al., or as an intrinsic part of the language itself, as in KernelGenius.

Whilst the problem of combining operators appears as a recurring theme in image processing DSLs, the issue of combining GPU kernels has also been addressed as a research topic in its own right. In the next section, we will review the current state of the art with regard to kernel fusion.

## 3.5 KERNEL FUSION

Our work on generating DSLs on SYCL was partially motivated by a need to investigate approaches to kernel fusion on SYCL. More specifically, we wished to explore whether SYCL's programming model was sufficient to enable the generation of fused kernels without the need for additional tools beyond the SYCL compiler itself.

Loop fusion is a compiler optimization technique in which two or more loops are combined into a single loop. This may result in reduced loop overhead. However, loop fusion does not always result in improved run-time performance. It may also lead to reduced data locality, and a subsequent degradation in performance. It is therefore sometimes desirable to split an existing loop into two or more loops. This is referred to as loop fission.

We can regard a GPU kernel as a function representing a loop body, evaluated in parallel across each element of the work grid. It is therefore reasonable to assume that we can also apply similar fusion and fission techniques to GPU kernels. However, fusion applied to a GPU kernel must additionally be constrained so as to not

introduce data dependencies that would violate the constraints of the kernel execution model described in section 2.4. Most notably this requires the maintenance of compatible grid layouts, the avoidance of data dependencies between work-groups and the correct use of synchronisation primitives where data dependencies between work-items are introduced.

Matsuzaki and Emoto (2009) suggest four possible approaches to implementing loop fusion:

- As an optimization inside a compiler.
- As a code generator or translator.
- Through a library which performs compile-time optimization.
- Through a library which performs runtime optimization.

Whilst Matsuzaki and Emoto suggested these approaches in the context of implementing fusion for parallel skeletons (Cole, 1988) targeting multi-core CPUs, all of their approaches remain valid in the context of heterogeneous systems.

G. Wang, Lin and Yi (2010) propose the application of fusion to independent kernels, with the goal of improving GPU utilization and improving power efficiency. However, this work does not attempt to fuse data dependent kernels. Instead they focus on identifying independent work which can be freely combined. Kernels are combined using three schemes: concatenating the kernel functions, effectively executing the two kernels sequentially; or appending the work-grids and using either the work-item or work-group identifier as the discriminant of a conditional statement, where each branch represents a single input kernel.

Fousek, Filipovic and Madzin (2011) detail a decomposition-fusion scheme for automatically generating fused CUDA kernels. Computational problems are expressed in terms of elementary functions which form a simple standalone DSL. Source-to-source compilation is then combined with a cost model in order to generate one or more CUDA kernels representing the complete computation. This approach has conceptual similarities to the approach which we will adopt in chapter 4. However, where Fousek, Filipovic and Madzin utilize a separate source-to-source compiler, our DSL is deeply embedded within C++. This gives us strong typing and tighter integration with host code. Our approach currently lacks a cost model. However, it could reasonably extended to include one through the use of C++11's constant expressions, and this would present an interesting avenue for future work.

With KFusion, Kiemele et al. (2013) describe a tool for performing fusion of OpenCL kernels. More specifically, KFusion aims to address fusion in the case where an application developer wishes to exploit libraries of kernels. Whilst the previous works have described the fusion of kernels themselves, modifying such kernels in the manner previously described implies that the invoking host code must also be modified. In KFusion, host and kernel source code are augmented with additional annotations to indicating data flow requirements. Both host and kernel code are then analysed and new source files synthesised. Unlike the preceding work by G. Wang, Lin and Yi, KFusion focuses on fusing data dependent kernels. The approach taken by KFusion is attractive in that most of the burden of complexity in adding additional annotations is placed on the library developer, and not the third-party application developer. However, KFusion also places extremely strong constraints on the structure and organisation of source code and has not been demonstrated in the more general case.

Filipovic and Benkner (2015) provide an analysis of kernel fusion techniques applied to NVIDIA and AMD GPUs, an Intel CPU and Xeon Phi. In this work, the fused kernels are generated manually. However, Filipovic and Benkner additionally explore a number of fusion strategies which cannot currently be expressed by our DSL, such as the fusion of non-data dependent kernels. For expressions composed solely of point-wise operators, our DSL will generate kernels similar to the advanced data-dependent case described by Filipovic and Benkner. The performance benefits of our automatically generated kernels appear to be consistent with the results presented in this work, which can be interpreted as further validation of our approach. Whilst Filipovic and Benkner do not find all forms of kernel fusion to be beneficial in all cases, it is consistently advantageous for data dependent kernels. This directly corresponds to our image processing use case where the application of a sequence of point-wise operators is a common pattern. For alternative application domains where kernels are commonly compute-bound, or with few data dependent kernels, such fusion techniques are less likely to deliver performance benefits.

Expression templates have previously been applied to the problem of loop fusion for single-threaded CPU execution (T.L. Veldhuizen and Jernigan, 1997) and multi-core parallel skeletons (Matsuzaki and Emoto, 2009). They have additionally been used to provide the host interface to OpenCL-based linear algebra libraries such as VexCL (Demidov, 2012) and ViennaCL (Rupp, Rudolf and Weinbub, 2010). However, prior to the release of SYCL, the lack of a C++ compiler for kernel code meant that these projects must either execute multiple elementary kernels, or construct an

OpenCL C kernel dynamically at runtime (Demidov et al., 2013). Bawidamann and Nehmeier (2011) describe one such approach to dynamically generating OpenCL C kernels from expression templates.

VexCL (Malcolm et al., 2012) and the OpenCL back-ends for Halide and ArrayFire (Malcolm et al., 2012) all generate kernels through run-time manipulation of OpenCL C strings. SYCL does not mandate online compilation support. Expression trees within our methodology are compile-time structures, with the structure of expressions represented within the C++ type system. Because of this, we avoid the difficult and error-prone task of translating DSL expressions into OpenCL C strings. Under our approach the burden of kernel generation is placed upon the offline SYCL compiler. This results in stronger integration with the C++ type system, and provides the practical benefit of leveraging the optimizations available in mature C++ compilers.

## 3.6 RAY TRACING ON HETEROGENEOUS SYSTEMS

Ray tracing (Whitted, 1979), and related algorithms such as path tracing (Kajiya, 1986), form an important class of computer graphics algorithms. These algorithms aim to generate images from a virtual scene by simulating the interaction of light with surfaces. These algorithms are attractive due to their conceptual simplicity and for the ability to simulate a wide range of visual phenomena such as motion blur, depth of field and shadow penumbrae within a single framework (Cook, Porter and Carpenter, 1984). The same core algorithmic principles have also been applied to problems from other fields such as acoustics and seismology (Cerveny, 1985; Krokstad, Strom and Sørsdal, 1968; Kulowski, 1985).

Chapter 6 describes RTKit, our new framework for exploring ray tracing performance on HSA. This framework builds upon our programming model and compiler for HSA, described in Chapter 5.

### 3.6.1 Hardware Accelerated Ray Tracing

As a computationally intensive task, significant effort has been dedicated to applying accelerator devices to ray tracing. Ray tracing algorithms can be regarded as embar-

rassingly parallel, with each ray or path evaluated independently of others. However, ray tracing algorithms can prove challenging for memory subsystems (Kopta et al., 2015).

Purcell et al. (2002) explored the mapping of ray tracing to early programmable graphics hardware. Their results, achieved through simulation, suggested that ray tracing on GPUs could be competitive with CPU implementations. In the same year, Carr, Hall and Hart (2002) demonstrated the use of a ATI Radeon 8500 GPU as a co-processor to perform ray-triangle intersection, whilst the host CPU was responsible for scheduling ray casting and accumulating the resulting radiance samples. This early work provides an example of a heterogeneous solution, but suffered due to limited bandwidth between the CPU and discrete GPU.

Subsequent to this early work, significant effort has been devoted to exploring approaches to optimizing ray tracing on a variety of hardware architectures. Many authors have addressed efficient ray tracing kernels for CPUs (Fuetterling et al., 2015; Wald, Johnson et al., 2017; Wald, Woop et al., 2014; Woop et al., 2013). Aila, Laine and Karras (Aila and Karras, 2010; Aila and Laine, 2009; Aila, Laine and Karras, 2012) have conducted extensive work exploring exploring various ray traversal algorithms and mappings of rays to work-items, focusing on NVIDIA GPU architectures. Similarly, Benthin et al. (2012) explore both single ray and Single Instruction, Multiple Data (SIMD) packet-based ray tracing on Xeon Phi coprocessors.

In recent years, several hardware vendors have produced ray tracing APIs for their platforms: OptiX (Parker et al., 2010; Robison, 2011) for NVIDIA GPUs, Embree (Wald, Woop et al., 2014; Woop et al., 2013) for x86 processors and Xeon Phi, and Radeon Rays (Advanced Micro Devices, 2016b) for AMD GPUs. In the embedded and mobile space, several researchers and hardware vendors have begun to explore hardware-based ray tracing (Keller et al., 2013; W. Lee, Shin et al., 2014; J. Nah et al., 2014; Spjut et al., 2012).

Finally, RTFact (Georgiev and Slusallek, 2008) describes an approach with strong similarities to that which we take in implementing RTKit. RTFact provides a library of building blocks suitable for prototyping ray tracing applications and algorithms, based on the use of C++ templates to provide support for packetized ray tracing. However, RTFact describes a pure CPU implementation, while we chose to focus on applying a similar approach to heterogeneous systems.

### 3.6.2 Heterogeneous and Hybrid Ray Tracing Systems

The works described in section 3.6.1 focus on providing efficient ray tracing implementations for a single class of Processing Unit (PU). Whilst a significant body of work exists exploring the performance of ray tracing on GPUs and dedicated accelerators such as the Xeon Phi, and a similar body of work on CPU ray tracing, less work has explored hybrid systems. This is in part due to the historical need for expensive memory copies between CPU and accelerator devices. This is one area where HSA, and shared memory System-on-Chip (SoC) designs, may offer potential solutions. In this section, we review a number of existing works that make use of heterogeneous or hybrid architectures to share the computational burden of ray tracing between CPU and GPU cores.

Brigade is a path tracer optimized for real-time gaming applications (Bikker and Schijndel, 2013). CPU cores manage the updating of dynamic state and rebuilding of acceleration structures to reflect motion of objects within the game. A GPU path tracer performs the bulk of rendering, using state computed by the CPU cores in the previous frame. Where CPU cores lack sufficient work maintaining game logic and computing acceleration structures to fully occupy them for the frame time slice, they may also be utilized to assist in path tracing.

A number of authors have combined conventional rasterization techniques with ray tracing. Using rasterization to compute intersection between primary rays and geometry is a common approach in these techniques, while some authors have further extended the approach to encompass shadow calculations.

Beck et al. (2005) proposed an early hybrid rendering technique combining rasterization and ray tracing. Shadow maps and geometry identification are computed using conventional GPU rasterization passes. The frame buffer is then transferred to the CPU, where reflections and refractions are computed through CPU ray tracing. A final GPU shading pass is then used to merge the output results.

C. Chen and Liu (2007) use rasterization to compute intersection points for primary rays, storing triangle identifiers and the barycentric coordinates of the intersection point into a frame buffer. All shading and tracing of further secondary rays is then continued using the CPU.

The work of Beck et al.; C. Chen and Liu can certainly be categorized as applying heterogeneous systems to ray tracing. Their approach differs from ours in that they

combine traditional GPU rasterization with CPU computation in a fixed configuration, rather than using the GPU for general purpose computation.

Sabino et al. (2012) propose a hybrid rendering pipeline which combines rasterization and ray tracing. The scene is first rasterized using a deferred shading technique (Deering et al., 1988). A G-buffer (Saito and Takahashi, 1990) is populated with intersection positions, geometry normals and the optical properties of materials at the intersection point. These values then form the inputs to a ray tracing stage based on OptiX (Parker et al., 2010) to compute secondary effects such as shadows and reflections.

Pajot et al. (2011) reformulate bidirectional path-tracing to make efficient use of both a CPU and GPU. In bidirectional path-tracing (Lafortune and Willems, 1993; Veach, 1997) paths are constructed by independently tracing paths from the camera and a light source. The vertices of these two independent paths are then connected and the radiance transfer between path vertices computed. Pajot et al. extend this by computing populations of light and camera paths using CPU cores, and then using a GPU to connect all light paths to all camera paths.

J.-H. Nah et al. (2010) describe MobiRT, a hybrid CPU-GPU ray tracer designed to overcome the limitations of mobile GPUs. Whilst they do describe an implementation which fits within the constraints of OpenGL ES (Khronos OpenGL ES Working Group, 2007), their evaluation was performed using an emulator on a desktop computer. Consequently, the performance results are unlikely to be reflective of performance on mobile hardware.

W. Lee, Hwang et al. (2015) describe a hardware architecture for hybrid rendering on mobile devices. The output image is subdivided into tiles and ray tracing or rasterization selectively applied at the granularity of whole tiles.

Whilst many of the works described above utilize both CPU cores and GPUs to accelerate ray tracing, none of them describe the use of heterogeneous processors with shared memory, such as we see with APUs and HSA. Concurrent with my own work, a recent work by Barringer, Andersson and Akenine-Möller (2016) on RayAccelerator explores similar concepts to RTKit, relying upon SVM to utilize an Intel multi-core CPU and integrated GPU to perform ray tracing. Similar to our approach in chapter 6, Barringer, Andersson and Akenine-Möller progressively explore the performance of single and packet-based CPU ray tracing, before adding SIMD shading and finally a hybrid scheduler enabling the utilization of both CPU and GPU cores.

Whilst Barringer, Andersson and Akenine-Möller bears similarities to our work in focusing on a heterogeneous processor with CPU and GPU cores and SVM support, their work relied upon OpenCL kernels to provide GPU acceleration, and describes a fixed implementation approach, as distinct from our goal of attempting to providing a toolkit to enable experimentation. Instead of focusing on a single configuration, our approach aims to make use of a single-source programming model and template meta-programming to enable the rapid exploration of the possible optimizations and mappings of tasks to PUs.

### 3.6.3 Acceleration Structures

The use of spatial acceleration structures is essential to achieving high throughput for ray-geometry intersection tests. A large body of work exists to address the data structures and traversal techniques necessary to achieve this. The optimal choice of data structures and traversal algorithms is typically hardware-specific. Consequently, multiple implementations may be required when targeting a heterogeneous system. We provide an evaluation of several such algorithms in chapter 6.

Santos et al. (2012) provides a performance analysis of eight kd-tree traversal algorithms for NVIDIA GPUs.

Wald, Slusallek et al. (2001) describe an approach to ray tracing on SIMD hardware. This technique utilizes Binary Space Partitioning (BSP) trees as acceleration structures. Multiple primary rays are combined into a packet corresponding to the SIMD-width of the hosting PU, with all rays within a packet being tested against the same BSP tree node in parallel, and following a common traversal order.

Multi-branching Bounding Volume Hierarchy (BVH) trees provide an alternative route to exploiting SIMD parallelism (Dammertz, Hanika and Keller, 2008; Ernst and Greiner, 2008; Viitanen et al., 2016; Wald, Benthin and Boulos, 2008). Rather than construct binary BVH trees, with a branching factor of two per node, multi-branching BVH trees feature a higher branching factor. SIMD instructions can then be used to evaluate a single ray against multiple Axis-Aligned Bounding Boxes (AABBs) encompassing child nodes in parallel. This is in contrast to the preceding approach described by Wald, Slusallek et al. (2001), where multiple rays are tested against a single child node at a time.

For intersection testing, acceleration trees are typically traversed using a depth-first search algorithm. For CPUs, this is commonly implemented by utilizing a stack to enable backtracking to previously visited nodes. This technique has also been utilized for GPU-based ray tracing (Aila and Laine, 2009; Zhou et al., 2008). Resource constraints require that such stacks are stored in external, off-chip memory.

Several authors have applied stackless tree traversal algorithms to ray tracing. T. Foley and Sugerman (2005) described two stackless algorithms for kd-trees: *kd-restart*, and *kd-backtrack*. These algorithms traverse a scene in depth order, updating the endpoint of the tested ray segment. This relies upon the fact that kd-trees divide space into non-overlapping regions and so are not applicable to BVH trees, and hence our work which relies upon BVH trees. However, Laine (2010) later provided a variant based on tracking traversal position through a *bit-trail*, which is applicable to BVH trees and is implemented within RTKit. An alternative approach to stackless tree traversal is to introduce links or *ropes* between neighbouring nodes (Havran, Bittner and Zára, 1998; MacDonald and Booth, 1990). Hapala et al. (2011) describes a link-based method that only requires the addition of a single pointer per tree node, unlike earlier works which suffered from high memory requirements. Barringer and Akenine-Möller (2013) described an approach to stackless traversal that requires no additional pointers or memory accesses compared to stack-based traversal. In addition to implementing Laine's *bit-trail* traversal algorithm, we provide implementations of both the Hapala and Barringer traversal algorithms within RTKit. Comparative performance results for these three tree traversal algorithms, along with a traditional stack-based approach can be found in section 6.3.11.

Laine, Karras and Aila (2013) detail the negative performance impact of combining complex material shaders with a ray geometry intersection logic in a single GPU kernel. Instead they suggest that the intersection and shading logic should be implemented as separate kernels. Whilst Laine, Karras and Aila focused on GPU kernel performance, we believe that separating shading and intersection and splitting the work between CPU and GPU cores in a heterogeneous system seems an approach worthy of further investigation.

## 3.7 DISCUSSION

This thesis describes three technical works: a DSL for image processing which builds upon SYCL; a C++14-based programming model for heterogeneous systems; and RTKit, a framework for exploring optimising ray tracing on heterogeneous systems.

In this chapter, we have reviewed prior works which relate to each of these fields.

We have reviewed the major software frameworks for computing on heterogeneous systems in section 3.2, with a primary focus on C++-based models.

In chapter 4, we will explore our work on implementing a deeply embedded DSL for image processing, using SYCL to provide hardware acceleration. This work serves an early evaluation of the SYCL programming model as the basis of higher-level programming models. It also demonstrates an approach to achieving loop fusion without the need for additional compilation machinery beyond the SYCL compiler. Finally, it acts as a supporting data point to our argument that C++ can provide a route to managing the complexity of heterogeneous systems, by enabling us to provide simple and familiar syntax to domain-experts without requiring machine expertise.

Many other authors have tackled the problem of exploiting parallelism and heterogeneous systems for image processing. Section 3.4 provides a review of these works.

Due to the high cost of accessing memory, many of the DSLs that we have described attempt to combine image processing operators into pipelines which eliminate intermediate memory accesses and reduce off-chip memory bandwidth requirements. Other authors outside the field of parallel image processing have addressed the issue of combining GPGPU kernels with the same goal. We review these more general approaches to kernel fusion in section 3.5.

Chapter 5 describes a compiler and programming model targeting HSA. HSA is a relative newcomer to the heterogeneous software ecosystem, and is positioned as a low-level foundational technology for developing parallel programming models, rather than as middleware targeted at application developers. Consequently, relatively few languages currently target HSA. We provide a review of several other works that target HSA in section 3.3.

Chapter 6 describes RTKit, our framework for ray tracing on heterogeneous systems, which builds upon our programming model for HSA. Ray tracing is a computa-

tionally intensive task, the acceleration of which has received considerable attention. However, relatively little work has addressed the optimization of ray tracing on shared memory heterogeneous systems such as APUs, and to our knowledge none has targeted HSA. We review selected works on accelerating ray tracing on heterogeneous systems in section 3.6.

# Part II

# SYCL

# 4 A DOMAIN SPECIFIC LANGUAGE FOR IMAGE PROCESSING IN SYCL

In motivating this thesis, we described three overarching themes:

- Providing early usage experience and validation to two new standards for heterogeneous computing: SYCL and Heterogeneous System Architecture (HSA)
- The development and application of new C++ programming models for heterogeneous computing.
- The use of those programming models to build domain-specific toolkits for problems in the field of visual computing.

SYCL was originally envisioned as a framework to allow for the development of higher-level C++ template libraries which could take advantage of OpenCL-based hardware acceleration. The primary goal of this work was to act as an early validation of SYCL's suitability to that goal. Using image processing as a use case, in this chapter we will explore how SYCL and template meta-programming can be used to implement a deeply embdded Domain Specific Language (DSL) which hides the complexity of OpenCL, and which can provide the benefits of kernel fusion by composing kernels from simpler primitives. This is accomplished without the need for additional external tooling beyond the SYCL compiler and runtime itself. In this way, we aim to demonstrate how SYCL can be used as a basis for constructing domain-specific libraries and abstractions which reduce the need for their users to directly address the complexities of heterogeneous systems.

We begin, in section 4.1, with an introduction to the problem of generating efficient image processing kernels. Expression templates are used to capture the syntax of statements written in our DSL. We will give a brief overview of this technique in section 4.2. Expression templates also enable us to delay expression evaluation. This delayed evaluation is necessary to enable us to provide an interface that mimics the familiar mathematical syntax common to image processing textbooks, whilst transparently performing parallel evaluation of these expressions within one or more SYCL kernels. Section 4.3 discusses the design and implementation of our DSL. We provide performance comparisons between our SYCL-based DSL, Halide and OpenCV in Sec-

tion 4.4, describe some limitations of our approach in Section 4.5 and end with some concluding comments in Section 4.6.

## 4.1 MOTIVATION

In exploring the implementation of a DSL on SYCL, we have three primary goals:

- To illustrate how high-level languages such as C++ can provide a route to managing the complexity of heterogeneous systems.
- To validate the suitability of SYCL as a foundation for building complex higher-level domain-specific C++ libraries.
- To demonstrate that SYCL can be utilized to generate efficient kernels through the composition of simpler primitives.

The programming of heterogeneous systems is a complex task, requiring understanding of the architectural characteristics of the various Processing Unit (PU) within a system, and the careful management of data movement between the host processor and accelerator devices. The use of C++ and template meta-programming enables our DSL to provide an interface based on mathematical syntax familiar to image processing experts, while leveraging SYCL to transparently provide support for heterogeneous PU and efficiently manage data movement. It is not necessary for end users of our DSL to understand SYCL, or the complexities of heterogeneous systems.

One of the stated goals of the SYCL specification is to provide a foundation which higher-level domain-specific C++ libraries can utilize to provide hardware acceleration on heterogeneous devices. CUDA is widely used in this context, but is specific to one hardware vendor's architectures. OpenCL can also be used in this context. A much wider range of hardware supports OpenCL than CUDA but the use of OpenCL C as the kernel language places additional burdens on developers by limiting code reuse and restricting portability between host and accelerator code.

By providing a shared-source model with few language restrictions and no non-standard language extensions, SYCL aims to provide a platform which is able to support a wider range of hardware than CUDA whilst presenting fewer barriers to C++ developers than OpenCL.

The work described in this chapter was conducted prior to be publication of the SYCL specification, and consequently served to provide feedback and validation on the SYCL specification to the SYCL working group, and on the ComputeCpp (Codeplay Software Ltd., 2016) SYCL implementation to Codeplay Software.

Whilst OpenCL and SYCL are able to provide a platform for targeting heterogeneous hardware, the use of both Application Programming Interfaces (APIs) requires a significant level of domain-specific knowledge.

Parallel primitives libraries such as ArrayFire (Malcolm et al., 2012), OpenCL Integrated Performance Primitives (OpenCLIPP) (Akhloufi and Campagna, 2014) and NVIDIA Performance Primitives (NPP) (NVIDIA Corporation, 2011) aim to provide collections of primitives to enable developers to exploit the performance of parallel processors without the burden of the domain knowledge typically required to program them. Other libraries such as OpenCV seek to provide higher-level domain-specific abstractions, hiding the underlying hardware acceleration behind opaque data structures.

Whilst these libraries are able to provide highly optimized implementations of algorithmic building blocks, they are necessarily unable to optimize all of the myriad permutations of these primitives that third party developers may wish to exploit, and so must focus on the most common use cases.

```
// Load an RGB image.
cv::Mat input = cv::imread(..., CV_LOAD_IMAGE_COLOR);

// Copy to an OpenCL compatible UMat
cv::UMat rgb;
input.copyTo(rgb);

// Convert to greyscale.
cv::UMat grey;
cv::cvtColor(rgb, grey, CV_BGR2GRAY);

// Calculate a thresholded binary image.
cv::UMat output;
cv::threshold(grey, output, 128, 255, cv::THRESH_BINARY);
```

**Listing 4-1**: Image Thresholding in OpenCV

We can illustrate this issue with a simple image processing example utilizing OpenCV. Thresholding is a simple image segmentation algorithm, which is commonly run on greyscale images or single channels of a multi-channel image representation such

as RGB. When an input source such as a camera provides images in a format other than greyscale, a colour space conversion may be required before the thresholding operator is run. Listing 4-1 provides an example of how this might be expressed in OpenCV.

**Figure 4-1:** Example of a Thresholding Operator

Whilst OpenCV provides both colour space conversion operators and a threshold operator, it does not provide a single operator representing the composite of these two primitive operators. Consequently the two primitive operators are executed in sequential order, with an intermediate result written out to memory. Where OpenCL is used to provide hardware acceleration for OpenCV, this results in the execution of two separate OpenCL kernels.

The OpenCL(Khronos OpenCL Working Group, 2012) programming model does not provide a mechanism to retain data on-chip between kernel invocations, and so reads and writes to DRAM must be used to transfer intermediate values between kernels. Load and stores to off-chip memory are typically significantly more expensive both in terms of energy and latency than arithmetic operations or corresponding operations to the on-chip register file in current Graphics Processing Units (GPUs).

This issue presents a fundamental challenge for library developers who aim to provide their users with APIs which provide high performance whilst minimizing power costs.

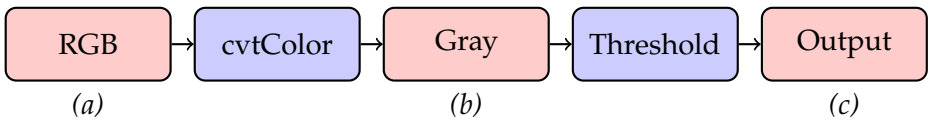



**Figure 4-2:** An Unfused Data Flow Graph for Colour Conversion and Thresholding Pipeline

The OpenCL buffers and kernels used by the OpenCV runtime to execute the example in listing 4-1 can be visualized as a data flow graph, as shown in figure 4-2.

In this graph, the buffers denoted *a* and *c* represent the input and output of our pipeline and cannot be eliminated. By contrast, the buffer *b* holds an intermediate result of the computation. If the colour conversion space kernel and the thresholding kernel can be merged into a single kernel then *b* could be eliminated without altering the observable behaviour of the pipeline. The result of this fusion is visualized in figure 4-3.


**Figure 4-3:** A Fused Data Flow Graph

The benefits of kernel fusion are well understood, and have been addressed by previous works (Filipovic, Madzin et al., 2015; Fousek, Filipovic and Madzin, 2011; Kiemele et al., 2013; Wahib and Maruyama, 2014; G. Wang, Lin and Yi, 2010). Whilst experienced OpenCL developers may choose to perform this fusion by hand, doing so requires both access to the kernel source code and expert knowledge of OpenCL.

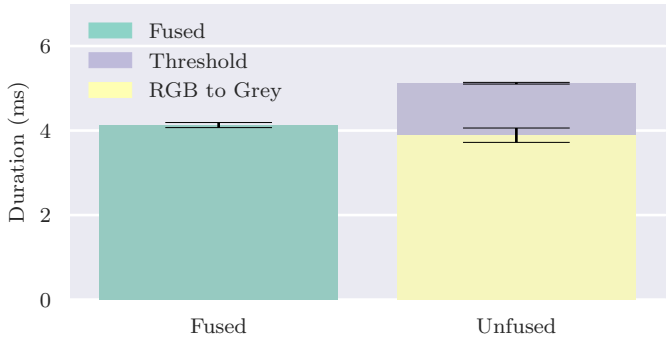



**Figure 4-4:** Performance Impact of Manually Fusing OpenCV's Colour Conversion and Thresholding Operators

We can explore the benefits of fusion by extracting the two OpenCL kernels which implement the colour space conversion and the threshold operator in OpenCV and manually fusing them. Table 4-1 and figure 4-4 illustrate the effect of this fusion. We can clearly see that fusing the two kernels has resulted in a 19% reduction in total kernel execution time.

Whilst manual fusion in this manner is often practical, it requires developers to have a familiarity with OpenCL. It is therefore desirable to automate this fusion.

| Implementation | Operator | Kernel(ms) | |
|---|---|---|---|
| | | $\overline{x}$ | $s$ |
| Fused | **Fused** | 4.13 | 0.06 |
| Unfused | RGB to Grey | 3.89 | 0.17 |
| | Threshold | 1.23 | 0.02 |
| | **Cumulative Unfused** | 5.11 | 0.17 |

**Table 4-1:** Performance Impact of Manually Fusing OpenCV's Colour Conversion and Thresholding Operators

Two approaches are typically taken to achieving this fusion. The first is to implement an offline tool, typically requiring additional annotation of source files to attempt to fuse kernels. The work of Kiemele et al. on KFusion (2013) is illustrative of this approach.

The second approach is to represent the kernels using a domain-specific language embedded within the host language, and to use the representation of this language to dynamically generate OpenCL kernels at runtime. Bawidamann and Nehmeier (2011) provide an example of how this can be implemented for simple matrix and vector operations. This is the approach taken by Halide and ArrayFire.

SYCL does support interoperability with OpenCL C kernels (Khronos OpenCL Working Group – SYCL subgroup, 2015, p. 135) and so this approach of dynamically generating OpenCL kernels at runtime could be utilized. However, this approach is non-idiomatic for SYCL. Additionally, it imposes a burden on the developer of a domain-specific language, by requiring the implementation of sufficient compilation functionality within the language support library.

An alternative approach is to statically generate the kernels at compile-time from expressions, placing the majority of the burden of compilation on an offline SYCL compiler, rather than requiring the implementation of such a framework within the runtime library. This is the approach that we take with our DSL.

## 4.2 EXPRESSION TEMPLATES

Expression templates were first introduced by T. Veldhuizen (1995), and Vandevoorde and Josuttis (2002).

An expression such as $x + y \cdot 99$ can be visualized as forming an abstract syntax tree. The values $x$, $y$ and 99 form the leaves of the tree, and are referred to as *terminals*, while the interior nodes represent *operators*. Figure 4-5 illustrates this.

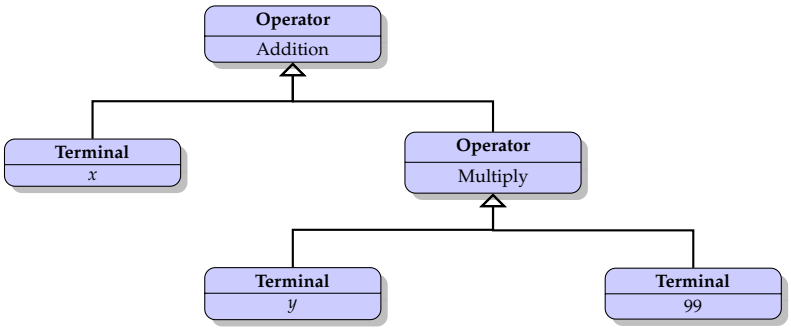



**Figure 4-5:** An Expression Tree for the Expression $x + y \cdot 99$

C++ templates and metaprogramming can be utilized to capture the semantics of an expression within the type system. We can define a templated type for terminal nodes as follows:

```
template<typename T> class Terminal;
```

Additionally, we can define a second type for binary operators:

```
enum Op { Add, /*...*/ };

template<Op op, typename LHSExpr, typename RHSExpr> class BinaryExpr;
```

These two types in combination can now be used to express the semantics of a simple computation. The type `BinaryExpr<Add, Terminal<float>, Terminal<float>>` now represents addition of two floating-point values. Whilst this type fully captures the syntax of an expression, building more complex expressions in this manner is syntactically unattractive. This situation can be improved by overloading existing C++ operators to manipulate our type-based expressions. If we overload **`operator+`**, then we can begin to write expressions that are simple to understand, but retain the property of capturing behaviour within the type system.

```
template<typename LHSExpr, typename RHSExpr>
auto operator+(LHSExpr lhs, RHSExpr rhs) {
  return BinaryExpr<Add, LHSExpr, RHSExpr){lhs, rhs};
}
```

Listing 4-2 demonstrates how this might then be used. It is important to note that the value assigned to `expr` in this example is an unevaluated representation of the expression, and not a concrete floating-point value. In this way, the usual expression evaluation which we might expect from C++ has been delayed.

```
1 Terminal<float> a;
2 Terminal<float> b;
3
4 auto expr = a + b;
```

**Listing 4-2:** A DSL Expression Embedded Within C++

By implementing a set of additional functions and operators, we can now begin to define a domain-specific language.

Expression templates have been widely used to implement high-performance linear algebra libraries. Blitz++ (T.L. Veldhuizen, 1998, 2000) provides an early example for Central Processing Unit (CPU)s. Other examples include uBLAS (Walter and Koch, 2002), Armadillo (Sanderson, 2010) and Eigen (Guennebaud, Jacob et al., 2010).

Several of these libraries target CUDA and OpenMP. Other authors have targeted OpenCL by constructing expression templates which internally generate OpenCL C strings, such as VexCL (Demidov, 2012), ViennaCL (Weinbub, Rupp and Selberherr, 2011) and ArrayFire (Malcolm et al., 2012).

No work to date has applied similar techniques to SYCL. Expression template libraries which target OpenCL commonly dynamically generate OpenCL C kernels at runtime. SYCL's offline compilation model means that a compile-time approach to kernel construction is desirable. This is both more idiomatic, and benefits from improved type-safety due to a stronger integration with the C++ type system. Whilst the dynamic generation of kernels is common for embedded DSLs targeting OpenCL, some libraries targeting CUDA have relied upon metaprogramming to enable compile-time generation of kernels (Guennebaud, Jacob et al., 2010). However, the SYCL programming model introduces additional constraints relating to the life cycle of accessors and buffer which increase complexity. These constraints are further described in section 4.3.3.

Having broadly described expression templates, in the next section we discuss the design and implementation of our DSL, and the issues that must be resolved in order to utilize SYCL to provide acceleration on heterogeneous devices.

## 4.3 DESIGN AND IMPLEMENTATION

We can divide the design and implementation of our DSL and runtime into three stages:

- Capturing the representation of an image processing expression.
- Transforming the captured expression into a form suitable for parallel execution on SYCL.
- Evaluation and execution of the expression on SYCL.

For the first stage, we make use of C++ expression templates to define the syntax and operators of our DSL. Our DSL aims to provide a syntax that closely follows the mathematical notation familiar to authors of image processing algorithms. Young, Gerbrands and Van Vliet (1998) describe a digital image $a[m,n]$ as a discretization of an analog two-dimensional function $a(x,y)$. They further describe many image processing operators in terms of this notation.

We adopt this same notation for describing image processing operators for the remainder of this chapter, and also use this notation as inspiration for the syntax of our DSL. Image processing pipelines are defined as mathematical expressions in our language, and then transformed into a form which both generates OpenCL kernels and schedules the necessary data movement to evaluate the expressions on an OpenCL accelerator.

Consider the simple example of computing the average intensity of a pair of images. This might be expressed mathematically as:

$$h(x,y) = \frac{f(x,y) + g(x,y)}{2}$$

Similarly, this expression might be implemented in our DSL as:

```
1 dsl::Image<RGB, uint8_t> f = ...;
2 dsl::Image<RGB, uint8_t> g = ...;
3 dsl::Image<RGB, uint8_t> h = ...;
4
5 auto expr = (h = (f + g) / 2.0f);
```

**Listing 4-3:** Calculating the Point-wise Average Intensity of Two Images in our DSL

As we can see, our DSL provides a terse, strongly typed syntax, which closely mirrors the syntax of mathematical form. It is important to note that the evaluation of the expression has not yet been performed in this example. We have merely captured a representation of the expression in the variable `expr`, and the contents of image $h$ have yet to be modified. Instead, evaluation is delayed until we explicitly evaluate the expression.

```
1 // Construct a SYCL queue to dispatch work to an accelerator.
2 cl::sycl::queue q;
3
4 // Perform asynchronous, parallel evaluation of expr.
5 dsl::eval(q, expr);
```

**Listing 4-4:** Expression Evaluation in our DSL

We evaluate our captured representation of the expression by passing the expression and a SYCL queue to an `eval()` function. This is demonstrated in listing 4-4. This function will transform the expression into a representation suitable for parallel evaluation within a SYCL kernel, and submit corresponding SYCL kernels for execution. Because the semantics of our expression are represented within the C++ type system, this process requires a series of compile-time transformations to be applied to the expression tree representation. A SYCL queue corresponds to a single OpenCL device, and so in this manner we can specify the PU used to evaluate the expression. This evaluation function will also cause the underlying SYCL runtime to generate OpenCL buffers and schedule the transfer of operands to the appropriate accelerator device where necessary.

This process also highlights a key difference between SYCL and the OpenCL C++ kernel language, previously described in section 2.6. An expression authored by a user of our DSL generates a unique type, which will ultimately be transformed into a correspondingly unique SYCL kernel which is dependent upon the expression type. Because the OpenCL C++ kernel language prohibits the use of C++ templates as kernel functions (Khronos OpenCL Working Group, 2017, p. 43), this technique cannot be directly applied to OpenCL using the C++ kernel language alone. Instead, dynamic compilation techniques similar to those previously described for frameworks which target OpenCL C are required.

### 4.3.1 Representing Images and Pixels

Operands within our DSL may represent *scalars*, *colours* or *images*.

We represent colours as partially specialized template types within C++, capturing both the colour space and the storage type used for the components. For example, an RGB colour value represented using 8-bits per channel can be expressed as `Colour<RGB, uint8_t>`. Capturing colour spaces in this manner provides us with additional type-safety by ensuring that arithmetic is only performed on values with consistent colour spaces. All C++ arithmetic types are treated as valid storage types.

Images are represented in a similar manner to colours, encapsulating both a colour space and a storage type within the image type. We currently treat all images as two-dimensional. However, support for three-dimensional images would be a trivial extension. The image types utilize a one-dimensional SYCL buffer to provide the internal storage for an image. SYCL also provides an image type, building upon OpenCL's images. These types allow an underlying OpenCL runtime to optimize data layout for performance and might seem a more obvious representation. Indeed, during development it was found that using SYCL images resulted in improved kernel throughput. However, they were also found to be significantly more expensive to construct and this resulted in an overall degradation in performance.

All C++ primitive arithmetic types are also regarded as valid scalar operands.

### 4.3.2 Representing Operators

We support point, neighbour and geometric operators. The implementation of an operator within our DSL consists of three parts. The first is a pure C++ functor providing the implementation of the operator. Listing 4-5 gives an example of a simple addition operator.

```
1 struct Add {
2   template<typename LHS, typename RHS>
3   auto operator()(const LHS& lhs, const RHS& rhs) {
4     return lhs + rhs;
5   }
6 };
```

**Listing 4-5:** Implementing an Addition Operator for our DSL

We must also provide a second pure functor which controls the evaluation of the operands of the expression. This functor accepts a sampling coordinate, optionally performs a transformation upon it, and then evaluates the operands. For point operators this evaluator can simply be an identity operator. Given a coordinate within the domain of an expression, this evaluator will evaluate each operand at the same coordinate and uses the resulting values to evaluate some operator. Listing 4-6 illustrates a simplified identity evaluator which only supports binary expressions.

```
template<typename Op>
struct Identity {
  template<typename LHSExpr, typename RHSExpr>
  auto operator()(LHSExpr& lhs, RHSExpr& rhs, Coordinate coord) {
    return Op()(lhs(coord), rhs(coord));
  }
};
```

**Listing 4-6:** Implementing an Identity Evaluator for our DSL

A more complex evaluator may perform some transformation on the input coordinate before using the modified coordinate to evaluate the operands. Geometric operators such as rotation and scaling can be accomplished in this manner. Evaluators may also sample their operands multiple times, with modified coordinates. This allows for the implementation of neighbourhood operators such as blur filters or edge detection kernels.

Each operator is then coupled with a function or overloaded C++ operator which provides the syntax for the operator within the DSL. Listing 4-7 demonstrates how an addition operator can be added to the syntax of our DSL by combining the addition functor from listing 4-5, the identity evaluator from listing 4-6 and an overloaded C++ operator.

```
template<typename LHSExpr, typename RHSExpr,
  typename = typename std::enable_if<is_expr<LHSExpr>::value &&
                                     is_expr<RHSExpr>::value>::type>
BinaryExpr<Add, Identity, LHSExpr, RHSExpr>
operator+(const LHSExpr& lhs, const RHSExpr& rhs) {
  return BinaryExpr<Add, Identity, LHSExpr, RHSExpr>{lhs, rhs};
}
```

**Listing 4-7:** Adding an Overloaded Operator to our DSL

We will now return to our example expression:

$$h(x,y) = \frac{f(x,y) + g(x,y)}{2}$$

We can now see that the code sample shown in listing 4-3 generates a complex templated C++ type. We can visualize this expression type as a graph, as shown in figure 4-6. From the figure we can see that the graph will be composed of a set of binary operator types representing the assignment, division and addition operators, and four terminals, which hold references to variables declared in the host language which must be used as the operands of the expression.

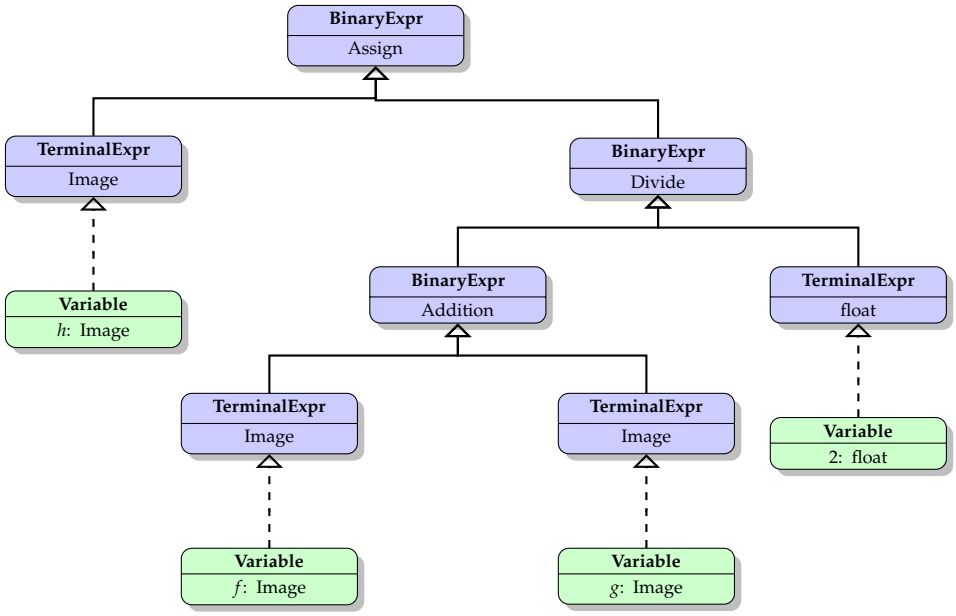



**Figure 4-6:** Host Representation of Binary Addition Expression Tree

### 4.3.3 Generating Pipelines

As discussed previously, the design of our DSL and runtime can be divided into three tasks: capturing a representation of the expression; transforming the expression into a form suitable for execution on SYCL; and expression evaluation where the transformed expression is finally executed. The preceding section has focused on how image expressions are represented and captured. This section will discuss the necessary transforms to manipulate these expressions into a form suitable for execution on SYCL.

The process of mapping from a user-authored expression to one or more OpenCL kernels consists of three compile-time transformations, expressed via the C++ template system. These transformations are applied automatically as part of the process of expression evaluation, without the need for input from end users.

1. Expression Fission
    - Splits the expression tree generated from a user-authored expression into one or more subexpressions, for performance or correctness reasons.
2. Operand Lowering
    - Operands referencing SYCL images or buffers must be replaced with operands containing SYCL accessors referencing the same resources.
3. Expression Evaluation
    - Generates and dispatches SYCL kernels per subexpression, based on the lowered expressions from the preceding transform.
    - Data dependencies are resolved automatically by the combination of accessors and the SYCL scheduler.

### 4.3.4 Expression Fission

Our goal is to generate efficient, fused kernels from multiple primitive operators. However, there are cases where it is either impossible or undesirable to fully fuse an expression into a single kernel. We must first address these cases before we begin the process of transforming a user-authored expression tree into a final kernel.

There are a number of reasons why it may be undesirable to blindly fuse a complete expression into a single kernel:

1. Correctness
    - The OpenCL execution does not allow for synchronization between workgroups within a single kernel.
2. Resource Constraints
    - Generating smaller kernels may reduce pressure on registers or shared memory, leading to improved occupancy.
3. Overlapping Memory Transfers

- SYCL must ensure all of the image operands of an expression are copied to the accelerator before executing the generated kernel. By splitting independent subexpressions, memory transfers can be overlapped with computation.

4. Common Subexpression Elimination
    - Whilst rematerialization (Briggs, K.D. Cooper and Torczon, 1992) is often an efficient approach for GPUs, in some cases it may prove more efficient to store the intermediate results.
5. Optimizing Separable Convolutions
    - Evaluating separable convolutions as two simpler convolutions results in significant bandwidth savings.

To accommodate these use cases, we introduce a new node type: `IntermediateExpr`. This node acts as a marker within our expression tree, indicating that we need to split the expression into two parts and generate two separate kernels. In splitting an expression into two parts, we will also need to generate a temporary, device-only image to store the intermediate result and schedule the execution of the two kernels in the correct order to avoid read after write data hazards.

```
1 Image<RGB, uint8_t> f = ...;
2 Image<RGB, uint8_t> g = ...;
3
4 auto expr = g = boxFilter2D<5>(f);
```

**Listing 4-8**: Calculating a 2D 5 x 5 Box Filter in our DSL

The transformation from a single expression tree containing a marker `IntermediateExpr` node to a pair of chained subexpressions without marker nodes requires the use of a compile-time transformation to fission our original expression tree into a pair of new expression trees. We can illustrate this process using a box filter. This is a separable kernel which can be expressed as a pair of one-dimensional blur filters. We can express a simple expression applying a two-dimensional box filter to an input image in our DSL as shown in listing 4-8. In authoring this simple expression, an end user simply applies a box filter function to an input image or expression, without consideration to how this expression might map to OpenCL kernels.

Listing 4-9 illustrates the implementation of the `boxFilter2D` function. As we can see, this constructs an instance of a complex templated type, composed of two unary expression nodes (representing 1D filters) separated by a marker node. This function

```
// This function constructs a sequence of expression tree nodes:
// BoxFilter1D_X<N> -> IntermediateExpr -> BoxFilter1D_Y<N>
template<uint32_t N, typename Expr>
auto boxFilter2D(const Expr& e) {
  static_assert(1 == N % 2,
                "Box filter size must be an odd number.");
  // The type of the subexpression that we wish to separate into a
  // separate kernel
  using SubExpr = UnaryExpr<Nop, BoxFilter1D_Y<N>, Expr>;

  // A concrete instance of that expression, which will include
  // references to the operands.
  auto filter_y_node = SubExpr{e};

  // A new expression with a marker node at the root.
  auto intermediate_node = IntermediateExpr<SubExpr>{
    filter_y
  };

  // The final full expression.
  return UnaryExpr<Nop,
                   BoxFilter1D_X<N>,
                   IntermediateExpr<SubExpr>>{intermediate};
}
```

**Listing 4-9:** Implementation of 2D Box Filters in our DSL

is provided as part of our DSL in this particular case. However similar alternatives might be provided by machine experts in the more general case.

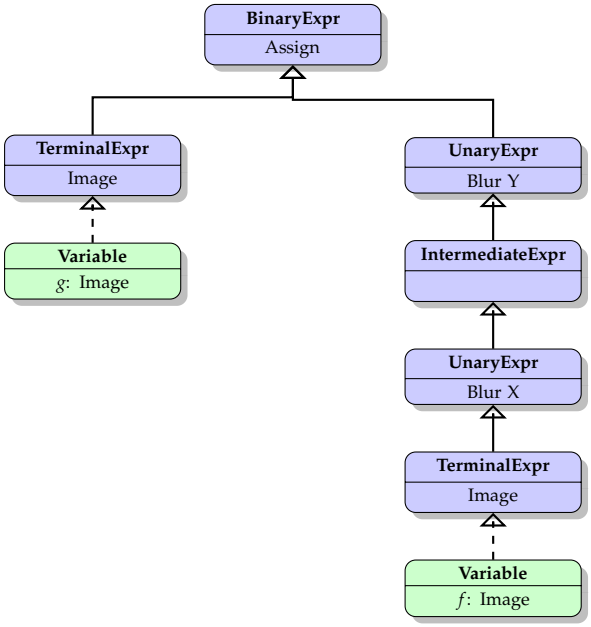



**Figure 4-7:** Untransformed Expression Tree for Separable Box Filter

The graph generated by this expression can be visualized as shown in figure 4-7. Here we can again see a pair of `UnaryExpr` nodes representing the horizontal and vertical blurs. These nodes are separated by an `IntermediateExpr` node, indicating that a temporary result should be stored in memory.

By applying our compile-time fission transformation, the expression is split into two subexpressions. The `IntermediateExpr` has been eliminated, replaced by a pair of `TerminalExpr` nodes sharing a common operand. This is illustrated in figure 4-8.We can now generate and execute kernels for these two subexpressions independently, provided we ensure that they are dispatched in the correct order.

This splitting of complex expressions into subexpressions is applied recursively until all `IntermediateExpr` marker nodes have been eliminated. Once an input expression has been decomposed into subexpressions in this manner, a further transformation must be applied to resolve restrictions imposed by SYCL's programming model, and constraints of the lifetime and scoping of SYCL's buffers and accessors. This requires the lowering of all operands that refer to SYCL buffers or images to accessors, and is described in section 4.3.5.

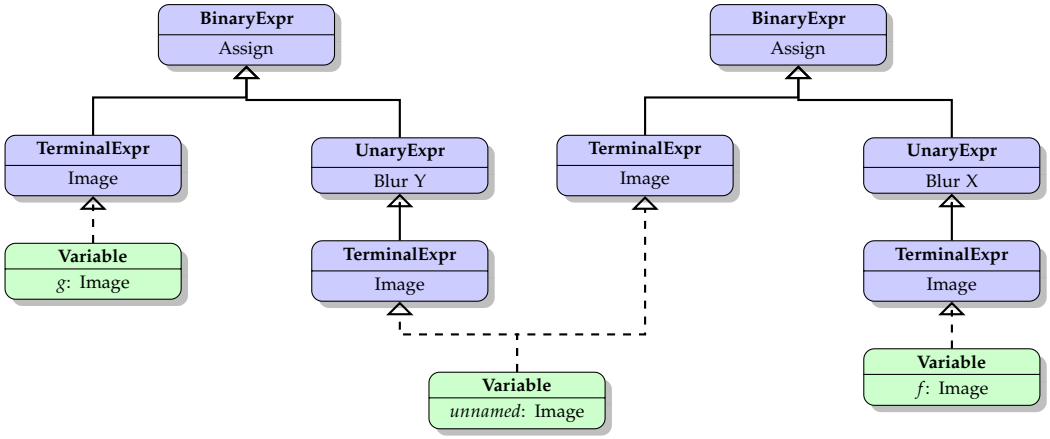



**Figure 4-8:** Transformed Expression Tree for Separable Box Filter

### 4.3.5 Operand Lowering

OpenCL provides two containers for data held in memory: images and buffers. For host code, SYCL provides a pair of corresponding types: `cl::sycl::image` and `cl::sycl::buffer`. However, instances of these types may not be directly accessed from within a SYCL kernel, and compound types which contain these types as members also violate SYCL's restrictions on valid data types with kernels. Consequently, an expression tree which contains references to SYCL's images or buffers cannot be evaluated directly within a SYCL kernel. These restrictions are ultimately rooted in the representation of memory objects in OpenCL 1.2.

Instead of accessing a `cl::sycl::buffer` directly, SYCL kernels gain indirect access through a `cl::sycl::accessor` type. These accessors serve the dual purpose of providing a type which matches SYCL's kernel language restrictions, and aiding the SYCL scheduler in identifying data dependencies between kernels.

Consequently, the expression type must be transformed from a form which references image terminals to a new form which references any image operands by indirecting through `cl::sycl::accessor` types.

There are some further constraints on how this transformation can be applied. A SYCL buffer may be freely constructed anywhere in the host code of an application, and is constructed without reference to a specific queue or accelerator device. Consequently, the image types used in our DSL may also be constructed with the same lack of constraints.

The construction of accessors is much more constrained. Accessors are transient objects which provide a link between a SYCL memory object such as a buffer or image and a specific kernel dispatch. The lifetime of an accessor is constrained to the scope of the function call operator of the command group lambda or function object.

From these constraints it follows that we must construct a new representation of our expression for evaluation within our SYCL kernel, and that the lifetime of this new representation is constrained in the same manner as that of SYCL's accessors.

The `cl::sycl::accessor` type also includes a compile-time constant parameter which indicates the required access permissions i.e. whether an accessor requires read-only, read-write or write-only access to the underlying buffer. The correct use of these parameters must be inferred contextually from the usage of terminals within an expression. For example, an image terminal on the left-hand side of an assignment operator must generate a writeable accessor.

In transforming expressions into a form compatible with SYCL's constraints on valid data types for use within kernel functions, each terminal in a user-authored expression is transformed into one of four possible device-compatible accessors.

Scalar and colour operands are values which evaluate uniformly across the domain of an image expression. These values also have relatively low storage requirements, with a maximum size of 128 bits for a 4-component floating-point RGBA colour. Terminals representing these values are transformed into `UniformExpr` nodes, which copy the operands by value. These nodes ultimately result in the operand value being passed into the generated SYCL as a kernel argument. Kernel arguments are variables in OpenCL's private address segment and typically consume GPU registers, so it is important that this approach is only applied to relatively small values.

Local, or neighbourhood, operators such as filters often include a set of weights used to perform a convolution with samples generated from the neighbourhood. Like the scalar and colour operands, these weights are uniform across the domain of the image expression. However, unlike scalar and colour operands, these sets of values tend to be larger. Using the previously described `UniformExpr` nodes for these values can result in extreme register pressure issues. On current Advanced Micro Devices (AMD) Graphics Core Next (GCN) GPUs, each Single Instruction, Multiple Data (SIMD) unit has a hard limit of 256 Vector General Purpose Registers (VGPRs) per work-item. In order to hide memory latency, each SIMD unit will attempt to keep up to 10 wavefronts active simultaneously, and these wavefronts must share register

resources. This leaves a budget of only 25 32-bit registers before register pressure begins to reduce the maximum number of wavefronts in flight. A general 5 x 5 filter might reasonably be represented by 25 floating-point weights, and consequently consume this entire budget before we allocate any resources for actual computation. To resolve this pressure we can shift these values to OpenCL's *constant* address segment. The runtime for our DSL does this transparently by copying these values into a SYCL buffer and transforming the terminal into a `ConstantBufferExpr` node, which holds a *constant* segment accessor.

Image operands are transformed into one of two possible nodes. Input images are transformed into `SampleImageExpr` nodes, which encompass a read-only SYCL accessor, while output images transformed into `WriteImageExpr` and generate discard-write accessors.

In order to perform these transformations, we define a set of templates which recursively perform compile-time pattern matching on an expression in order to replace the terminals of the expression with a new set of terminals. We can see the result of this transformation by comparing the original representation of our expression in figure 4-6 with a transformed representation as shown in figure 4-9.

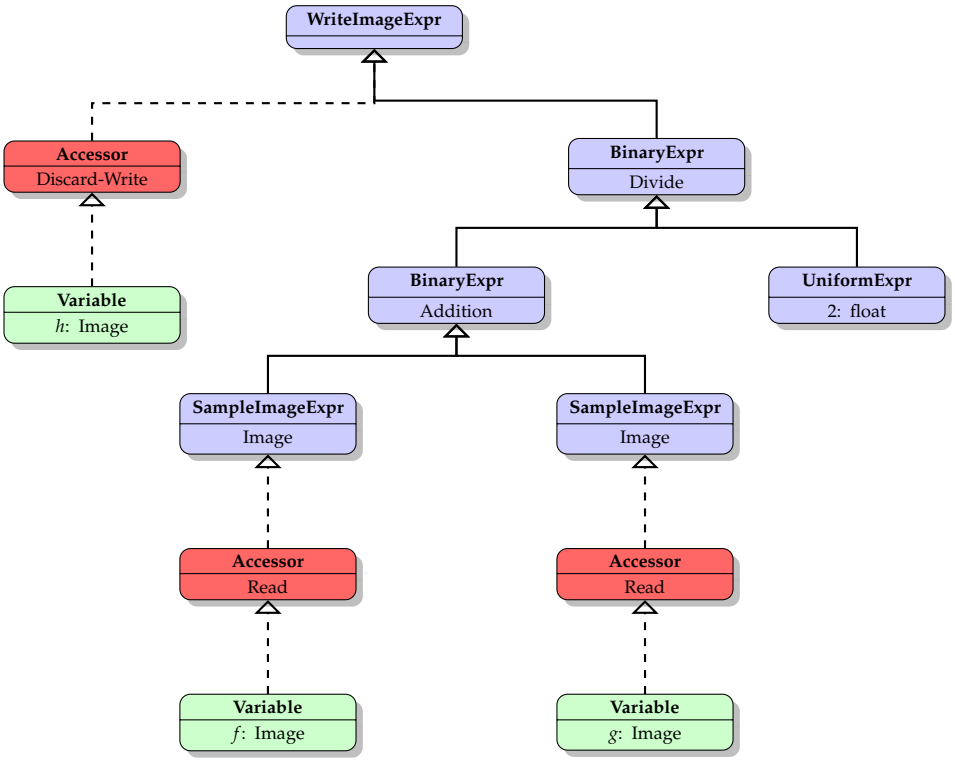



**Figure 4-9:** Device Representation of Binary Addition Expression Tree

We can see that two of the image terminals have been replaced with `SampleImageExpr` nodes, and associated read-only SYCL accessors have been generated. The remaining image terminal has been replaced with a `WriteImageExpr` node, along with a discard-write accessor. Finally, the floating-point divisor terminal has been transformed into a `UniformExpr` node. This performs a by-value copy of the divisor.

### 4.3.6 Expression Evaluation

Final expression evaluation takes place within one or more SYCL kernels, and therefore is performed in parallel on an OpenCL accelerator.

The output of the preceding sequence of transformations is an ordered list of one or more pipeline objects. Each pipeline is a C++ function object, designed to adhere to all of SYCL's restrictions on kernel code. All of the expression operands have been captured as member variables of this pipeline object. Scalars and colour values have been copied by-value, while images and buffers of constants are captured indirectly via accessors. Each pipeline object is parameterized by a single argument: a two-dimensional coordinate. These objects can therefore be captured in a SYCL kernel lambda function and evaluated for each pixel of the output image.

Listing 4-10 illustrates how parallel evaluation of an expression tree can be achieved. The variable `expr` represents an expression tree that was declared at *host scope*. Within the SYCL command-group handler this expression is lowered into a form suitable for use within a SYCL kernel, as described in section 4.3.3. The expression tree is then referenced within the kernel lambda function, and will therefore be copied to the OpenCL device.

A work-grid is created which maps one OpenCL work-item to a single pixel in the output image. Each work-item is then able to calculate a normalized two-dimensional coordinate and derivatives. These coordinates then be used to evaluate the pipeline object for each pixel in the output image.

Kernels within an OpenCL program must be uniquely named. Unfortunately lambda functions in C++ are anonymous types, and so cannot be used to provide a kernel name that is consistent in cases where different compilers are used for host and device code. SYCL resolves this by requiring a unique type name to be provided as a template argument to the `parallel_for()` method which represents a kernel dispatch. This type is then used to generate the kernel name. The use of the C++

```
template<typename Expr>
void eval(cl::sycl::queue queue, size_t height, size_t width,
          Expr&& expr)
{
  // Command-group handler to perform kernel dispatch.
  auto handler = [&](cl::sycl::handler& cgh) {
    // Construct a pipeline object from an expression.
    // This generates a new type, with all references to host-only
    // types replaced by accessors.
    auto pipeline = make_pipe(cgh, expr);

    // Enqueue a parallel_for task, using the pipeline type to
    // provide a unique name the kernel.
    cgh.parallel_for<decltype(pipeline)>(
        cl::sycl::nd_range<1>(cl::sycl::range<1>(width*height),
                              cl::sycl::range<1>(64)),
                                  [=](cl::sycl::nd_item<1> idx) {
      // Compute a normalized image coordinate and derivatives.
      float x = idx.get_global(0) % width;
      float y = idx.get_global(0) / width;

      Coordinate coord{x / width, y / height,
                       1.0f / width, 1.0f / height};

      // Evaluate the pipeline for the calculated coordinate.
      pipeline(coord);
    });
  };

  // Submit the command-group to the accelerator queue.
  queue.submit(handler);
}
```

**Listing 4-10:** SYCL Kernel for Parallel Evaluation of DSL Expressions

type system to represent expression trees provides the useful benefit of providing unique types for each expression. These types are used to uniquely name the kernels generated for expressions, as shown on line 14 of listing 4-10.

## 4.4 EVALUATION

In this section, we evaluate the performance of several image processing primitives in OpenCV, Halide and our DSL, demonstrating comparable kernel performance for individual primitives. Having excluded the performance of individual primitives as a contributing factor, we go on to demonstrate significant performance benefits to implementing more complex expressions in our DSL where they can be executed as a single kernel, as opposed to the more traditional multiple kernel approach adopted by OpenCV..

### 4.4.1 Evaluation Objectives

We evaluate OpenCV, Halide and our SYCL-based DSL across a number of benchmarks. OpenCV is a widely used image processing library. It acts as an exemplar of the traditional approach to providing an OpenCL backend to a domain specific library, providing a selection of hand-written OpenCL kernels. This is hidden behind a backend agnostic API. OpenCV contains no compilation machinery, and lacks support for kernel fusion. As an exemplar of the traditional approach to implementing a domain-specific library for OpenCL, we regard OpenCV as the baseline over which we aim to demonstrate performance benefits.

By contrast, Halide represents a state of the art embedded DSL, with a strong focus on support for exploring the impact of a wide range of approaches to scheduling image processing code. This implicitly includes support for kernel fusion. Halide's flexibility comes at the cost of requiring the developers to implement considerable compilation machinery within the library. Halide lacks the wide range of pre-written kernels provided by OpenCV, focusing instead on providing the tools to provide developers to author their own kernels.

In this evaluation, we will aim to demonstrate that our SYCL-based DSL provides similar performance benefits to Halide, particularly in relation to kernel fusion, des-

pite greatly reduced implementation effort. This reduced implementation effort comes at the cost of reduced flexibility, and we make no claims regarding our DSL achieving equivalent flexibility.

In particular, we aim to:

- Demonstrate that SYCL does not impose undue performance overheads beyond a more traditional OpenCL application, as exemplified by OpenCV.
- To validate the suitability of SYCL as a foundation for building complex higher C++ libraries.
- To demonstrate that SYCL can be utilized to generate efficient kernels through the composition of simpler primitives.

These objectives address two of the core contributions of this thesis, as described in section 1.3. Firstly, we aim to demonstrate that our DSL is able to provide the performance benefits of kernel fusion to image processing users without requiring machine expertise on their part. Secondly, this work acts as validation that the SYCL specification, and ComputeCpp implementation, are functionally sound, fit for purpose and do not impose undue performance overheads.

### 4.4.2 Evaluation Plan

In this section we describe our evaluation process. We present a series of benchmarks designed to illustrate a range of capabilities of our DSL, or to illustrate tradeoffs compared to alternative approaches. All of our benchmarks share a common structure. In each case, a small image processing pipeline chosen to illustrate a specific property is implemented in OpenCV, Halide and our SYCL-based DSL.

For each benchmark, two sets of measurements are taken. The first measure is the total execution duration of each pipeline. This measurement includes the cost of constructing the image representations in each API, memory copies to and from the GPU and kernel execution. We load benchmark images into arrays in system Dynamic Random-Access Memory (DRAM) and preallocate storage for output images outside of the timing loop, whilst the construction of API specific storage such as SYCL buffers is included in the host timings. This provides us with a proxy for the total cost of processing a single image.

We also measure kernel execution times. These measurements are generated by using an interposer library between the high-level runtime libraries (OpenCV, Halide and SYCL) and the OpenCL runtime. This enables us to transparently enable profiling support without modifying the high-level runtimes.

The kernel execution measurements were calculated separately from the host execution measurements. This ensures that the overhead of the interposer library, and the additional API costs of enabling profiling, do not contaminate the host measurements.

Due to the manner in which OpenCL exposes performance counters, it is only possible to query per-kernel execution durations, rather than collecting timings amortized over multiple iterations of a timing loop. For consistency, we also compute host timings per individual loop iteration. This approach excludes loop overhead from our results, but is potentially sensitive to timer precision. We address this in section 4.4.4. Due to high variance in kernel execution times, we execute each benchmark 100 times. The first iteration of each kernel carries additional setup and compilation overhead, resulting in disproportionately high execution times. Therefore we exclude this first iteration from our results. In addition to countering OpenCL's kernel setup overhead, this also ensures that Halide's one off Just In Time (JIT) compilation costs are excluded from our measurements.

After verifying that the precision of timers in our evaluation system will not have a significant negative impact on our results (section 4.4.4), we begin our evaluation by focusing on point-wise operators.

In order to establish a baseline for comparing performance, we will begin by evaluating the performance of the simple primitive operators that will act as the building blocks for constructing larger pipelines. This both allows us to eliminate differences in the performance of basic primitives as a factor in subsequent benchmarks, and provides simple examples for the comparison of host-side overheads. Benchmarks for a selection of primitive operators can be found in sections 4.4.5 and 4.4.6.

After establishing a baseline for the performance of component primitives, we will investigate the performance of our first pipeline composed from our primitives. In section 4.4.7 we evaluate a simple image brightening example. For Halide and our DSL, this example requires the construction of a simple pipeline from the components previously evaluated in sections 4.4.5 and 4.4.6. By contrast, a dedicated implementation of the pipeline already exists in OpenCV.

The preceding example represents a strength of OpenCV, utilizing a common operation for which OpenCV has a dedicated, hand-written implementation. However, libraries such as OpenCV necessarily cannot provide implementations of all possible image processing operators. In section 4.4.8 we will explore an image desaturation example. This is an example for which OpenCV must fall back to executing multiple GPU kernels, whilst Halide and our DSL are able to generate a single fused kernel. This example provides a demonstration of the performance benefits of kernel fusion.

The preceding examples are all point-wise operators, acting as a one-to-one mapping between input and output pixels. Section 4.4.9 provides a downsampling example, where multiple input pixels are accumulated into a single output pixel.

Our final sets of benchmarks address neighbourhood operators. Here we follow a similar approach to that adopted for point-wise operators. We first evaluate the performance of a simple Gaussian blur across our three example platforms (section 4.4.10) and then follow with an example based on unsharp masking, which demonstrates the combination of kernel fusion and neighbourhood operators.

### 4.4.3 Experimental Setup

Our results were evaluated using the integrated GPU in an AMD A10-7850K processor. The evaluation uses OpenCV 3.1, Halide 2016/04/27 and Codeplay Software's ComputeCpp 16.07 SYCL implementation. These represent the latest released versions of each framework at the time of evaluation. All three of these libraries support the use of the same OpenCL implementation for hardware acceleration. We use the OpenCL driver from the AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) v3.0 (version 1800.8).

For all of the benchmarks evaluated in this section, the inputs consist of 13.5 megapixel images. These are typically RGB images with 8-bits per component unless otherwise noted.

### 4.4.4 Validating Timers

Many of the benchmarks presented in this chapter require the measurement of execution times on the order of 10 ms to 100 ms. Given the relatively small time periods

involved, it is important to verify that the results are not negatively impacted by the resolution of the timestamps, or the update frequency of the clocks.

For the purposes of the benchmarks in this chapter, we are concerned with two separate timers. For kernel timing, we utilize the clock exposed by the OpenCL runtime via the profiling events. For the OpenCL implementation used in this evaluation, this clock has both a resolution and reported update frequency of 1 ns.

For timings observed by the host processor, such as those measuring API overheads, we make use of the C++ high resolution clock. On our evaluation system, this clock also reports timestamps at nanosecond resolution. However, this clock does not allow us to query its update frequency. We therefore measured this empirically, by repeatedly sampling pairs of timestamps. Table 4-2 summarizes the result of sampling 1 billion pairs of timestamps.

| **Timer** | **Duration (ns)** | | | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | Min | Max |
| Host CPU | 17.00 | 21.60 | 13 | 129000 |

Table 4-2

From table 4-2, we can conclude that whilst the C++ high resolution clock, and associated API overhead does offer us somewhat reduced accuracy when compared to the OpenCL clocks, it still provides timestamps at approximately five to six orders of magnitude higher than the duration of the benchmarks that we are evaluating. Whilst the vast majority of timestamp pairs fall into the range of 10 ns to 30 ns, we do observe a relatively small number of timestamp pairs that represent significantly larger durations (50,000 ns to 130,000 ns). We believe these correspond to process context switches. However, even these larger durations are not large enough to significantly impact our benchmarks.

### 4.4.5 Assignment Operators

We begin our performance evaluation by exploring the performance of the simplest expressions in our DSL. By defining expressions which consist solely of a single assignment operator we can fill an image with a constant colour or copy one image into another.

```
// Declare an input and output images.
dsl::Image<RGB, uint8_t> g(...);
dsl::Image<RGB, uint8_t> f(...);

// Declare a SYCL queue.
cl::sycl::queue q;

// Assign a scalar to every pixel in f.
dsl::eval(q, f = 128);

// Assign a colour to every pixel in f.
dsl::eval(q, f = Colour<RGB, uint8_t>(128, 256, 0));

// Copy image g to f.
dsl::eval(q, f = g);
```

**Listing 4-11:** Scalar, Colour and Whole Image Assignment Operators in our DSL

Listing 4-11 illustrates the syntax to fill an image with a colour or scalar value, and how to copy an image using an expression in our DSL. The performance of each of these operations relative to implementations in Halide and OpenCV are shown in figure 4-10 and table 4-3.

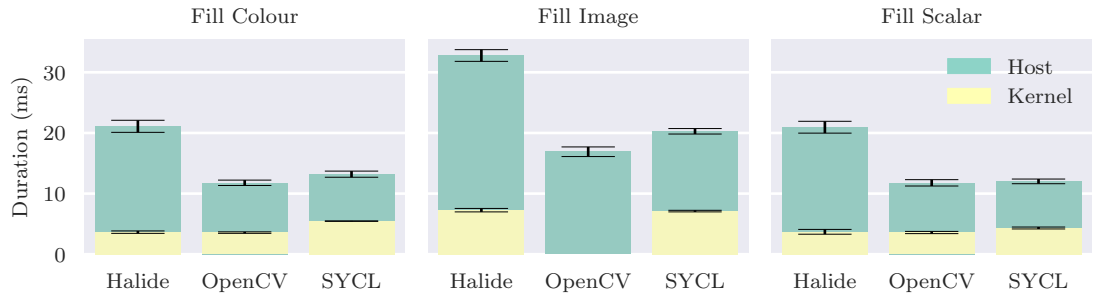



**Figure 4-10:** Assignment Operator Performance in Halide, OpenCV and our DSL

These simple examples highlight a number of common factors which we will see repeated across many of the subsequent benchmarks.

The first feature of note is the relative efficiency of host API usage. Our evaluation reports both host and kernel execution times. Halide, OpenCV and SYCL all attempt to intelligently manage resource allocation and memory transfers within their respective runtimes. In each case, we have endeavoured to apply equivalent, efficient usage of each runtime. However, we opted to treat each runtime as a black box

| Runtime | Kernel (ms) | | Total (ms) | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| **Assign Scalar:** $f(x, y) = 128$ | | | | |
| Halide | 3.70 | 0.40 | 20.95 | 0.98 |
| OpenCV | 3.58 | 0.18 | 11.78 | 0.52 |
| SYCL | 4.33 | 0.16 | 12.02 | 0.39 |
| **Assign Colour:** $f(x, y) = \text{RGB}(128, 256, 0)$ | | | | |
| Halide | 3.64 | 0.20 | 21.10 | 1.00 |
| OpenCV | 3.57 | 0.12 | 11.78 | 0.44 |
| SYCL | 5.46 | 0.04 | 13.20 | 0.51 |
| **Assign Image:** $f(x, y) = g(x, y)$ | | | | |
| Halide | 7.27 | 0.29 | 32.78 | 0.97 |
| OpenCV | | | 16.90 | 0.79 |
| SYCL | 7.12 | 0.13 | 20.28 | 0.45 |

**Table 4-3:** Assignment Operator Performance in Halide, OpenCV and our DSL

and not apply any optimizations that would require modification of the runtimes themselves.

In principle, each example is performing equivalent work in these examples. In the scalar and colour fill benchmarks, a single output image is allocated and a kernel is executed to populate each pixel. In the image fill benchmark, an input image must also be allocated and each pixel sampled and copied to the output image. We might therefore expect broadly equivalent usage of the underlying OpenCL runtime from all three runtimes. In practice, we observe very similar usage of the OpenCL API from both OpenCV and our DSL via ComputeCpp. However, Halide's usage of the OpenCL API is less efficient, particularly with respect to memory copies and to the blocking `clFinish` API call. These inefficiencies are internal to the Halide implementation and performance could presumably be improved with further optimization from the Halide authors. The code to manage data movement and kernel dispatch within Halide is generated dynamically through JIT compilation, and so is challenging to inspect in more detail. However, we note that a comparatively high host-side overhead appears to be a consistent factor for Halide across all of our benchmarks.

The second feature of note relates to the scalar and colour assignment kernels. For these kernels, both Halide and OpenCV are able to dynamically generate kernels which outperform our statically compiled kernel. However, the approaches they

adopt differ. OpenCV is able to generate vectorised stores, while Halide generates scalar stores. However, Halide is able to emit the scalar or colour component values directly into the kernel as constant literals. This technique cannot be easily replicated in our DSL whilst retaining the offline compilation behaviour of our approach. Both OpenCV and Halide's approaches give excellent kernel performance.

Finally, OpenCV does not utilise an element-wise GPU kernel to perform an image copy. Instead, a simple memory copy is performed. This makes direct comparison difficult in this case. However, we do see equivalent performance from kernels written in Halide and our DSL.

### 4.4.6 Primitives

We continue our performance evaluation by investigating the performance of simple expressions composed of single primitives from our DSL. Our goal here is to illustrate that the performance achieved using our DSL in later examples derives from the generation of efficient kernels by composing these expressions together, and not simply from the implementation of individual primitives. The performance of these primitives was evaluated both for 8-bit integer and 32-bit floating-point RGB images. The results of these evaluations can be found in table 4-4 and figures 4-11 and 4-12.

The performance of all of these operators is dominated by the cost of memory transfers, due to extremely low arithmetic intensity. We see differing results for 8-bit integer and 32-bit floating-point based representations.

For the 8-bit representations, primitives in our DSL give comparable performance to the primitives expressed in Halide. OpenCV fares somewhat better in most cases. This can be attributed to two related factors. The operators profiled here perform identical computations across all three channels of an RGB image. In our DSL and Halide, we map each individual pixel in the domain of our expression to a single OpenCL work-item. When operating on RGB images, as we do in these benchmarks, each work-item evaluates the operator across the 3 components of RGB colours. The kernels in OpenCV are optimized according to the observation that component-wise operators can be safely parallelized and vectorized on a per-component rather than per-colour basis. OpenCV assigns four contiguous components per work-item, rather than the three components used in our Halide and SYCL implementation. This enables reads to be aligned to 4 byte boundaries and improved Arithmetic Logic Unit (ALU) usage. The notable exception to this is the bitwise and operator, where

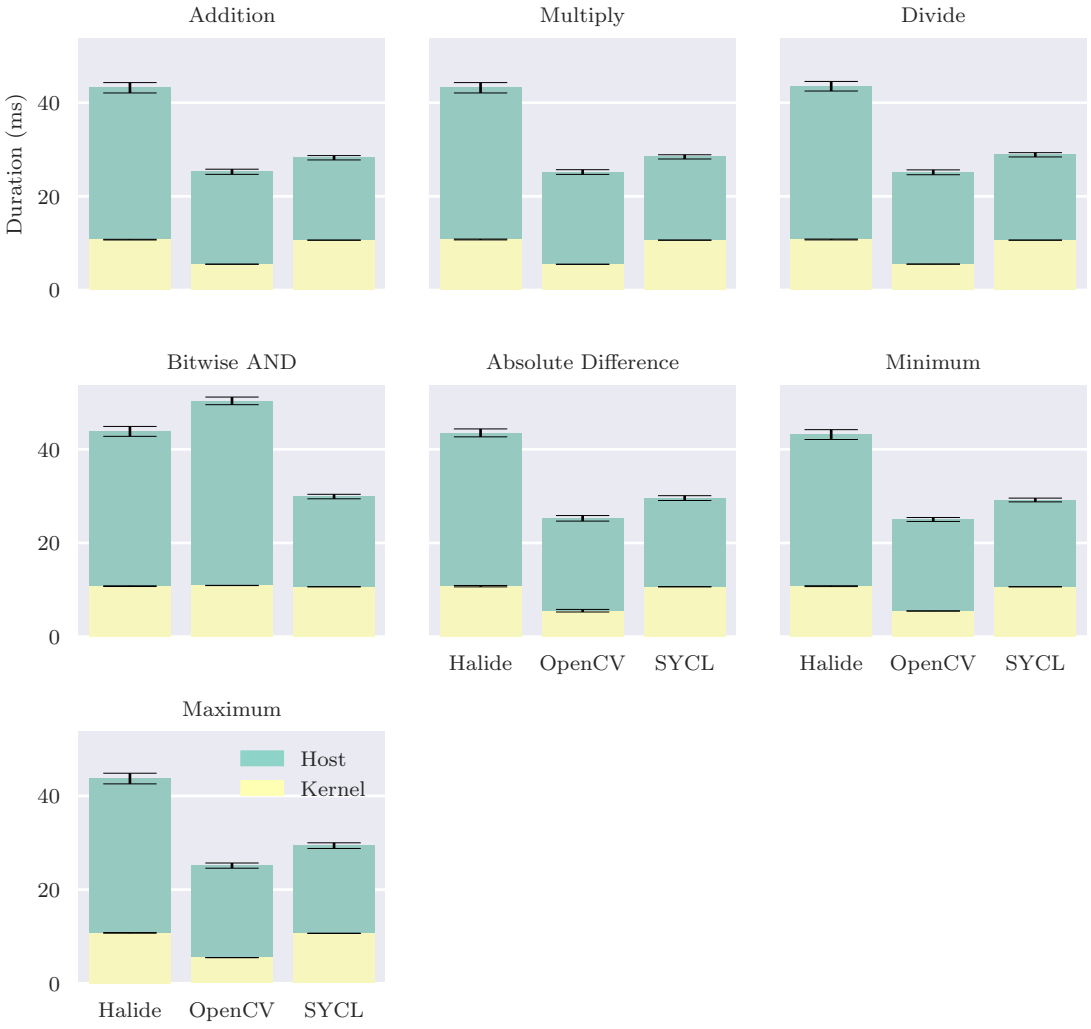



**Figure 4-11:** Primitive Operator Performance in Halide, OpenCV and our DSL: 8-bit Integer RGB

| Runtime | RGB u8 | | | | RGB f32 | | | |
|---|---|---|---|---|---|---|---|---|
| | Kernel (ms) | | Total (ms) | | Kernel (ms) | | Total (ms) | |
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| **Addition:** $f(x,y) = g(x,y) + h(x,y)$ | | | | | | | | |
| Halide | 10.76 | 0.07 | 43.20 | 1.11 | 30.32 | 0.16 | 193.25 | 4.86 |
| OpenCV | 5.51 | 0.02 | 25.25 | 0.55 | 22.96 | 0.50 | 85.39 | 0.84 |
| SYCL | 10.65 | 0.00 | 28.25 | 0.46 | 22.44 | 0.07 | 82.13 | 0.35 |
| **Multiply:** $f(x,y) = g(x,y) \cdot h(x,y)$ | | | | | | | | |
| Halide | 10.79 | 0.09 | 43.20 | 1.11 | 30.31 | 0.18 | 192.33 | 4.83 |
| OpenCV | 5.49 | 0.02 | 25.20 | 0.50 | 22.92 | 0.56 | 85.34 | 0.90 |
| SYCL | 10.65 | 0.00 | 28.43 | 0.45 | 22.45 | 0.07 | 82.25 | 0.39 |
| **Divide:** $f(x,y) = \frac{g(x,y)}{h(x,y)}$ | | | | | | | | |
| Halide | 10.79 | 0.09 | 43.52 | 1.02 | 29.63 | 0.28 | 192.88 | 4.69 |
| OpenCV | 5.53 | 0.02 | 25.14 | 0.52 | 21.62 | 0.02 | 84.09 | 0.71 |
| SYCL | 10.65 | 0.00 | 28.88 | 0.46 | 22.41 | 0.06 | 82.74 | 0.52 |
| **Bitwise And:** $f(x,y) = g(x,y)$ & $h(x,y)$ | | | | | | | | |
| Halide | 10.77 | 0.08 | 43.83 | 1.06 | | | | |
| OpenCV | 10.92 | 0.02 | 50.35 | 0.82 | | | | |
| SYCL | 10.65 | 0.00 | 29.92 | 0.48 | | | | |
| **Absolute Difference** $f(x,y) = \text{abs\_diff}(g(x,y), h(x,y))$ | | | | | | | | |
| Halide | 10.77 | 0.15 | 43.51 | 0.85 | 30.01 | 0.17 | 192.64 | 4.57 |
| OpenCV | 5.53 | 0.27 | 25.26 | 0.59 | 22.64 | 0.63 | 85.00 | 1.00 |
| SYCL | 10.65 | 0.00 | 29.59 | 0.51 | 22.45 | 0.08 | 83.86 | 0.70 |
| **Minimum:** $f(x,y) = \min(g(x,y), h(x,y))$ | | | | | | | | |
| Halide | 10.77 | 0.09 | 43.15 | 1.05 | 30.32 | 0.17 | 192.54 | 4.93 |
| OpenCV | 5.47 | 0.02 | 25.02 | 0.42 | 22.97 | 0.51 | 85.00 | 0.75 |
| SYCL | 10.65 | 0.00 | 29.18 | 0.40 | 22.44 | 0.06 | 82.95 | 0.48 |
| **Maximum:** $f(x,y) = \max(g(x,y), h(x,y))$ | | | | | | | | |
| Halide | 10.75 | 0.07 | 43.71 | 1.14 | 30.30 | 0.16 | 192.38 | 4.35 |
| OpenCV | 5.47 | 0.02 | 25.12 | 0.55 | 23.02 | 0.47 | 86.00 | 0.80 |
| SYCL | 10.65 | 0.03 | 29.38 | 0.60 | 22.44 | 0.06 | 83.45 | 0.49 |

**Table 4-4:** Primitive Operator Performance in Halide, OpenCV and our DSL

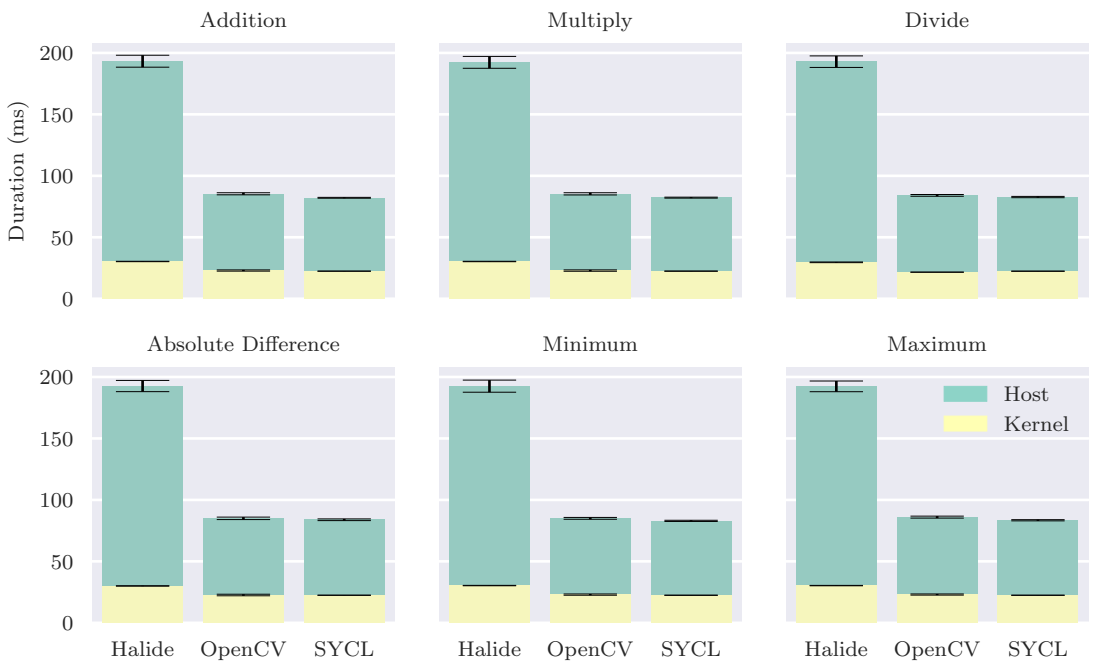



**Figure 4-12:** Primitive Operator Performance in Halide, OpenCV and our DSL: 32-bit Floating-point RGB

OpenCV appears to adopt a different structure compared to its other kernels, and to only achieve equivalent performance to our kernels, coupled with increased host overhead.

By authoring a SYCL kernel which replicates the access pattern utilized by OpenCV, we were able to achieve equivalent performance. However, this optimization is not valid for all operators, and so could not be applied universally within our DSL. Instead, our DSL could be extended to detect operators where such an optimization is applicable by adding a type trait to component-wise operators and modifying the kernel schedule accordingly where applicable. This remains future work. This tradeoff between generality and more focused optimizations is one of the disadvantages of our approach.

For 32-bit floating-point representations, OpenCV does not apply the previously described optimization, and so we achieve equivalent performance throughout. The kernels generated by Halide appear to be approximately 30% slower than either the OpenCV or SYCL kernels. Halide generates Static Single Assignment (SSA)-like OpenCL C kernels without obvious flaws. These are then passed to the OpenCL C compiler within the OpenCL runtime, which is a black box to us, and which appears

to generate less optimal code for these particular kernels. Halide also exhibits increased host overhead for the 32-bit floating point benchmarks over that exhibited by the 8-bit benchmarks. In section 4.4.5 we noted that host overhead is difficult to diagnose in Halide due to the use of JIT compilation. However, one plausible hypothesis is that this overhead relates to the initialization or copying of image buffers, which would be larger for the 32-bit benchmarks.

In summary, Halide, OpenCV and our SYCL-based DSL deliver broadly equivalent kernel performance. The notable exceptions are that the hand-written OpenCV kernels are able to exploit a more efficient mapping of work on 8-bit data types, and Halide suffers from some modest inefficiencies on floating-point data types. This more efficient mapping for 8-bit types is a strength of OpenCV's approach, allowing for specialized kernels with less need for generality. As anticipated, Halide and our SYCL-based approach do not deliver significant performance performance advantages for these primitive kernels. Consequently, any performance benefits observed in later benchmarks cannot be simply be attributed to faster implementations of basic primitives. We also note that SYCL has performed well in terms of host overhead in all cases.

Thus far, we have only addressed the relative performance of kernels which perform a single, primitive operation. However, the strength of both Halide and our DSL-based approach lies in allowing developers to express larger sections of an algorithm in a single kernel. This gives greater opportunities for compiler optimizations. Specifically, many intermediate calculations which would require off-chip memory operations in OpenCV are able to remain in registers with our approach. This results in significant improvements in both performance and energy consumption. In the next section, we will begin to explore operators composed from multiple simpler operations. We will begin in section 4.4.7 with an image brightening example for which OpenCV has a single specialized kernel, and follow this in section 4.4.8 with a desaturation example, for which OpenCV must execute multiple kernels.

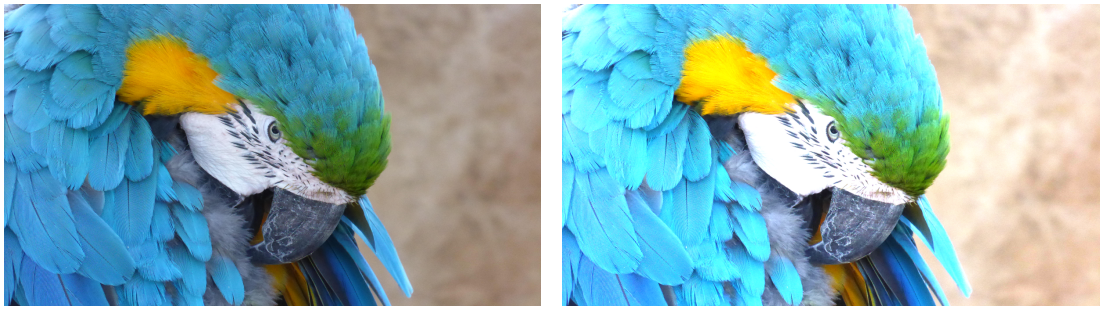


**Figure 4-13:** Example of Image Brightening Operator

### 4.4.7 Image Brightening

This benchmark combines several point operators to form a simple pipeline. Each pixel is scaled in intensity by 1.5×, and then clamped to a maximum brightness.

$$f(x,y) = min(g(x,y) \cdot 1.5, 255) \tag{2}$$

This example represents a best case for OpenCV. The operations are uniform across the channels of an RGB colour, which makes it suitable for the application of the vectorization optimization described in the preceding section. It is also an operator for which OpenCV provides a single kernel, effectively eliminating the need to store any intermediate values to memory. OpenCV's `convertTo()` function provides a single, hand-optimized kernel which supports scaling and clamps output values to the maximum saturated colour. This results in excellent performance for this single operation, at the cost of flexibility. For example, we cannot adjust the maximum intensity value to clamp to while using this function. Listing 4-12 illustrates the usage of OpenCV to provide such an operator.

```
1 cv::UMat g(height, width, CV_8UC3);
2 cv::UMat f(height, width, CV_8UC3);
3 g.convertTo(f, CV_8UC3, 1.5);
```

**Listing 4-12:** An Image Brightening Pipeline in OpenCV

Halide and our DSL take a different approach, requiring us to construct an expression representing our desired operator from a series of simpler operators. The runtime libraries will then generate a single OpenCL kernel from this expression. This allows for greater flexibility than the approach taken by OpenCV. The format of these

expressions in both Halide and our DSL closely mirrors the mathematical notation presented in equation (2)

Listing 4-13 illustrates an implementation of this brightening pipeline, taken from tutorial 2 in the Halide distribution. A key strength of Halide is in allowing developers to rapidly experiment with different approaches to scheduling computations for the purposes of hardware specific optimization. We exclude GPU scheduling optimizations from this example for brevity, however they were applied when generating all of the presented results.

```
// Declare an input image.
Halide::Image<uint8_t> g;

// Declare the brightening function.
Halide::Var x, y, c;
Halide::Func brighten;
brighten(x, y, c) = Halide::cast<uint8_t>(min(g(x, y, c) * 1.5f, 255));

// Apply the function to generate output.
Halide::Image<uint8_t> f = brighten.realize(g.width(),
                                            g.height(),
                                            g.channels());
```

**Listing 4-13:** An Image Brightening Pipeline in Halide

Listing 4-14 illustrates an equivalent example implemented in our DSL.

```
// Declare an input and output images.
dsl::Image<RGB, uint8_t> g(...);
dsl::Image<RGB, uint8_t> f(...);

// Declare the brightening function and apply to generate output.
cl::sycl::queue q;
dsl::eval(q, f = min(g * 1.5f, 255));
```

**Listing 4-14:** An Image Brightening Pipeline in our DSL

The performance of the image brightening operator can be seen in figure 4-14 and table 4-5. Our DSL achieves a fractional kernel performance improvement over Halide. However, OpenCV significantly out performs both Halide and our DSL. This performance can be attributed to a more efficient scheduling approach adopted by OpenCV. Similarly to the preceding results, we accrue a slightly smaller performance overhead from our DSL runtime and SYCL than that found in OpenCV, suggesting

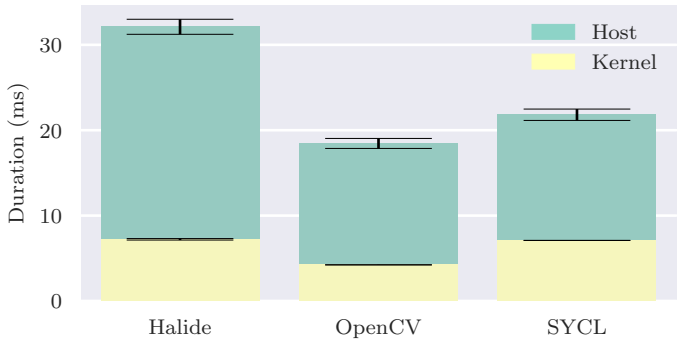



**Figure 4-14:** Image Brightening Performance in Halide, OpenCV and our DSL

that the scheduler in ComputeCpp is performing well and that the API is imposing comparable costs.

| **Runtime** | **Kernel (ms)** | | **Total (ms)** | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| Halide | 7.22 | 0.09 | 32.12 | 0.88 |
| OpenCV | 4.23 | 0.01 | 18.45 | 0.59 |
| SYCL | 7.09 | 0.00 | 21.81 | 0.67 |

**Table 4-5:** Image Brightening Performance in Halide, OpenCV and our DSL

### 4.4.8 Image Desaturation

In the image brightening example in section 4.4.7, OpenCV was able to offer excellent performance due to the existence of a specific, hand-optimized kernel dedicated to the required task. In this next example, we will consider a use case where OpenCV lacks such a kernel. Instead, a pipeline must be constructed by executing several kernels. By contrast, Halide and our DSL are able to generate a single fused kernel. In doing so, unnecessary intermediate memory operations are eliminated, resulting in improved kernel performance for both Halide and our DSL.

We use a desaturation operator to demonstrate this issue. An input image, $f(x, y)$, is transformed from the RGB colour space into HSV. The saturation channel is then zero'ed and the inverse colour space transformation is applied to return the image to the RGB colour space.

Listing 4-15 illustrates how a desaturation pipeline can be implemented in our DSL.

**Figure 4-15:** Example of Desaturation Operator

```
// Declare an input and output images.
dsl::Image<RGB, uint8_t> g(...);
dsl::Image<RGB, uint8_t> f(...);

// Define a mask to zero the saturation channel.
auto mask = Colour<HSV, uint8_t>(1, 0, 1));

// Convert input image g to the HSV colour space, zero the saturation
// channel and convert back to RGB colour space.
cl::sycl::queue q;
dsl::eval(q, f = convert<RGB>(convert<HSV>(g) * mask);
```

**Listing 4-15:** An Image Desaturation Pipeline in our DSL

Figure 4-16 and table 4-6 show the performance of the primitive operators required to implement this desaturation operator. We can clearly see that for each individual operator there are only small performance differences been Halide, OpenCV and our DSL.

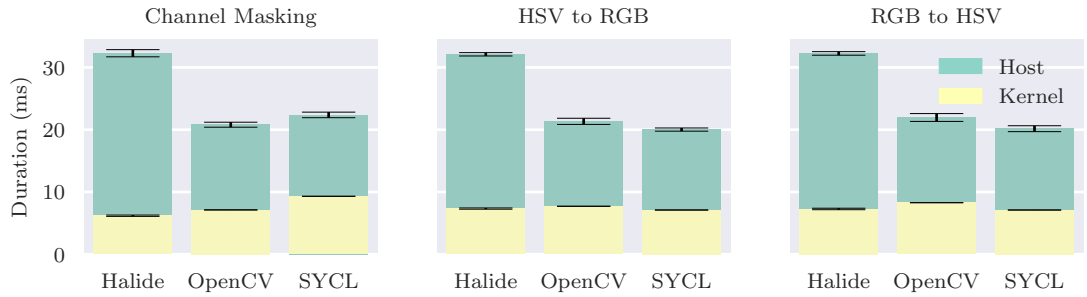



**Figure 4-16:** Component Operator Performance for Desaturation Pipeline in Halide, OpenCV and our DSL

| **Runtime** | **Operator** | **Kernel(ms)** | |
|---|---|---|---|
| | | $\overline{x}$ | $s$ |
| Halide | Channel Masking | 6.18 | 0.11 |
| | HSV to RGB | 7.33 | 0.10 |
| | RGB to HSV | 7.27 | 0.12 |
| | **Cumulative** | 20.78 | 0.19 |
| OpenCV | Channel Masking | 7.12 | 0.01 |
| | HSV to RGB | 7.70 | 0.00 |
| | RGB to HSV | 8.28 | 0.01 |
| | **Cumulative** | 23.10 | 0.01 |
| SYCL | Channel Masking | 9.30 | 0.01 |
| | HSV to RGB | 7.10 | 0.00 |
| | RGB to HSV | 7.10 | 0.00 |
| | **Cumulative** | 23.50 | 0.01 |

**Table 4-6:** Component Operator Performance for Desaturation Pipeline in Halide, OpenCV and our DSL

When we execute the complete pipeline this situation alters significantly. Figure 4-17 and table 4-7 show execution times for the complete desaturation pipeline. Both Halide and our DSL are able to generate a single kernel for this pipeline. This results in greatly decreased kernel execution times when compared to the cumulative execution times of component kernels shown in the table 4-6. These kernels have very low arithmetic intensity and so the execution time of these kernels is dominated by

the cost of data movement. By combining three kernels into one, both Halide and our DSL succeed in reducing total kernel execution time to approximately $30 - 35\%$, whilst the total kernel execution time observed for OpenCV remains close to the cumulative times calculated in table 4-6.

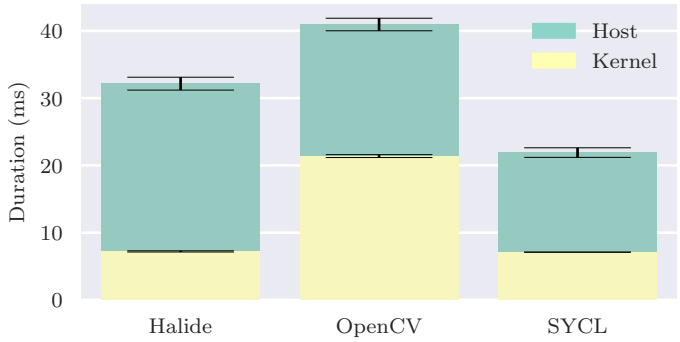



**Figure 4-17:** Full Pipeline Performance for Desaturation Pipeline in Halide, OpenCV and our DSL

In addition to decreasing the total execution time of kernels, fusion has allowed for the elimination of temporary memory buffers. We can see the impact of this by comparing the host execution costs for SYCL and OpenCV in figure 4-17. In the preceding benchmarks, the cost of host execution has been comparable for OpenCV and SYCL, with only small efficiency gains favouring SYCL. However, in this case we see a wider gap. Our DSL spends 13.24 ms on host execution, while OpenCV requires 17.95 ms. This additional 35% overhead can be attributed to the construction of additional buffers and the increased scheduling overhead of executing three kernels instead of one.

| **Runtime** | **Kernel (ms)** | | **Total (ms)** | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| Halide | 7.22 | 0.09 | 32.15 | 0.96 |
| OpenCV | 21.38 | 0.21 | 40.94 | 0.93 |
| SYCL | 7.10 | 0.00 | 21.91 | 0.71 |

**Table 4-7:** Full Pipeline Performance for Desaturation Pipeline in Halide, OpenCV and our DSL

This example serves to highlight some key strengths of our approach. By utilizing expression templates, we achieve significant performance improvements over implementations which are unable to perform fusion such as OpenCV. We also fractionally

exceed the kernel performance of Halide, which represents the current state of the art for image processing languages.

### 4.4.9 Downsampling

All of the operators that we have explored thus far have been evaluated with identically sized input and output images, and therefore a simple 1-to-1 mapping between pixels. However, our DSL also supports geometric transformations such as scaling and rotation by performing transformations on the sampling coordinates used to evaluate the expressions.

We demonstrate this by implementing downsampling of an input image to 75% size in OpenCV, Halide and our DSL.

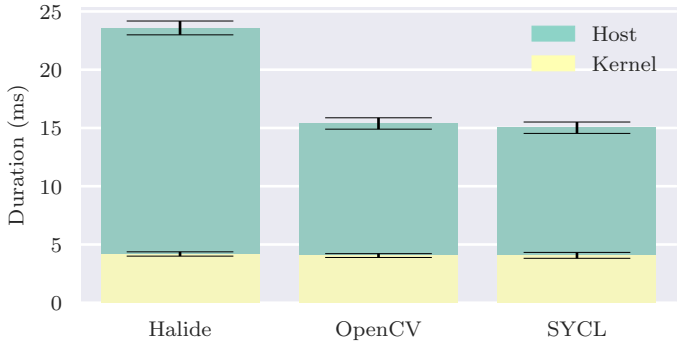



**Figure 4-18:** Downsampling Operator Performance in Halide, OpenCV and our DSL

| **Runtime** | **Kernel (ms)** | | **Total (ms)** | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| Halide | 4.18 | 0.18 | 23.59 | 0.59 |
| OpenCV | 4.05 | 0.16 | 15.39 | 0.49 |
| SYCL | 4.07 | 0.25 | 15.02 | 0.49 |

**Table 4-8:** Downsampling Operator Performance in Halide, OpenCV and our DSL

As we can clearly see from figure 4-18 and table 4-8, all 3 runtimes achieve comparable kernel performance, while OpenCV and ComputeCpp also achieve equivalent host overhead.

### 4.4.10 Filtering and Neighbourhood Operators

The preceding examples have all consisted of expressions constructed solely from point operators. In the following examples we will now examine neighbourhood operators. This example serves two purposes. Firstly, it acts as a demonstration of support for operators other than simple point operators within our DSL. Secondly, it provides an example of a case where improved performance can be achieved by breaking an expression into two parts and executing them as separate kernels.

We will begin by considering a simple mean or box filter, applied over a $(2 \cdot m + 1)$ x $(2 \cdot n + 1)$ neighbourhood around each pixel:

$$g(x,y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} \frac{f(x+i, y+j)}{(2 \cdot m + 1) \cdot (2 \cdot n + 1)}$$

We could implement a box filter as a double nested loop, iterating over the surrounding neighbourhood independently for each pixel. This will generate a correct result, but requires $(2 \cdot m + 1) \cdot (2 \cdot n + 1)$ samples from the input image $f$ for each pixel.

An alternative approach is to separate the box filter into a pair of motion blurs:

$$h(x,y) = \sum_{i=-m}^{m} \frac{f(x+i, y)}{(2 \cdot m + 1)}$$
$$g(x,y) = \sum_{j=-n}^{n} \frac{h(x, y+j)}{(2 \cdot n + 1)}$$

This reduces the number of samples per pixel to $(2 \cdot m + 1) + (2 \cdot n + 1)$. However, taking this approach does incur some additional costs. We must now allocate additional image storage for the intermediate image $h$. Because the OpenCL execution model does not provide a method of work-group synchronization which guarantees independent forward progress between work-groups, we must now execute two kernels instead of one.

Figure 4-19 and table 4-9 show the result of applying three sizes of box filter using our DSL. From these figures we can see that splitting the box filter into two parts results in performance improvements even on the smallest filter. Consequently, we

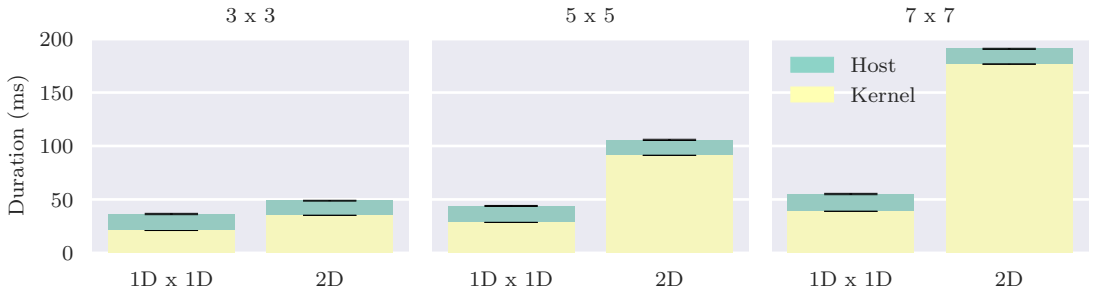



**Figure 4-19:** Separated and Combined Box Filter Performance in our DSL

choose to split all expressions involving separable convolutions into subexpressions, as described in section 4.3.3.

| **Filter Size** | **1D x 1D** | | | | **2D** | | | |
|---|---|---|---|---|---|---|---|---|
| | **Kernel (ms)** | | **Total (ms)** | | **Kernel (ms)** | | **Total (ms)** | |
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| 3 x 3 | 20.92 | 0.02 | 36.36 | 0.63 | 35.14 | 0.01 | 48.82 | 0.40 |
| 5 x 5 | 28.52 | 0.02 | 43.90 | 0.47 | 91.26 | 0.02 | 105.72 | 0.57 |
| 7 x 7 | 38.90 | 0.02 | 55.05 | 0.63 | 176.61 | 0.23 | 190.90 | 0.52 |

**Table 4-9:** Separated and Combined Box Filter Performance in our DSL

In the following section, we will explore the performance of a pipeline which requires a Gaussian blur as a component. Therefore, we present results for a Gaussian blur applied over a 5 by 5 window. Here we can again see comparable kernel performance for Halide and our DSL, with Halide achieving a fractional advantage.

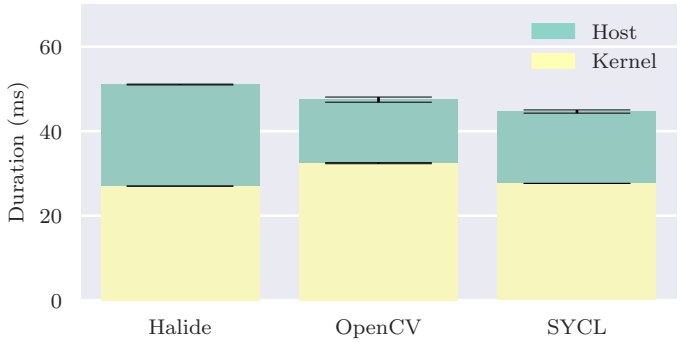



**Figure 4-20:** Gaussian Blur Operator Performance in Halide, OpenCV and our DSL

| Runtime | Kernel (ms) | | Total (ms) | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| Halide | 27.01 | 0.00 | 51.04 | 0.10 |
| OpenCV | 32.43 | 0.13 | $47.00 \times 10^{467,367}$ | 0.61 |
| SYCL | 27.65 | 0.02 | 44.65 | 0.38 |

**Table 4-10:** Gaussian Blur Operator Performance in Halide, OpenCV and our DSL

### 4.4.11 Unsharp Mask

Unsharp mask is a traditional image processing technique used for sharpening edges in images, first introduced by Schreiber (1970). Edges are high frequency changes in colour intensity. Given an image, $f(x, y)$, we can isolate the low frequency features by applying a Gaussian filter, $G$. By subtracting these low frequency features from the source image, we are able to extract the high frequency features.

$$g(x, y) = f(x, y) - G(f(x, y))$$

By adding some weighted portion of the high frequency features to the source image, we can strengthen the appearance of edges.

$$h(x, y) = f(x, y) + g(x, y) \cdot k$$

The colour space used to represent an image can introduce visual artifacts when sharpened in this manner. The RGB colour space components contain both intensity and colour information. As a consequence, using sharpening filters on RGB images results in modifications to both luminance and chrominance. This can result in false-colour edges. One approach to reduce these artifacts is to transform the input image to a colour space which represents luminosity and chromaticity separately, such as CIE*LAB or YCrCb, and to perform the sharpening solely on the luminance channel (Wirth and Nikitenko, 2010).

We present results for both an unsharp mask operation performed in RGB space, and for the same operation combined with a transformation to and from YCrCb space.

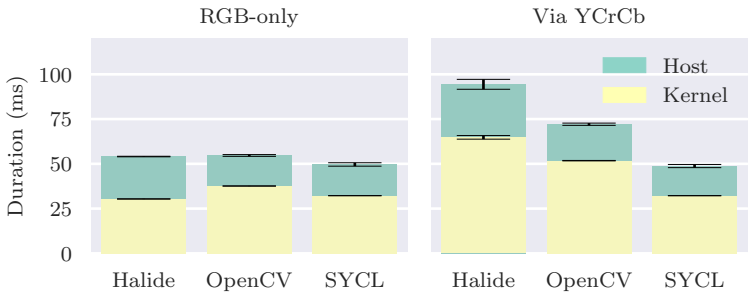



**Figure 4-21:** Unsharp Mask Pipeline Performance in Halide, OpenCV and our DSL

| **Runtime** | **Kernel (ms)** | | **Total (ms)** | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| **RGB-only** | | | | |
| Halide | 30.38 | 0.08 | 54.08 | 0.15 |
| OpenCV | 37.61 | 0.07 | 54.68 | 0.49 |
| SYCL | 32.22 | 0.01 | 49.65 | 0.96 |
| **Via YCrCb** | | | | |
| Halide | 64.79 | 1.01 | 94.43 | 2.77 |
| OpenCV | 51.77 | 0.13 | 72.12 | 0.63 |
| SYCL | 32.22 | 0.01 | 48.77 | 0.89 |

**Table 4-11:** Unsharp Mask Pipeline Performance in Halide, OpenCV and our DSL

We can make several observations from these results. The RGB variant of unsharp mask can be implemented using only two operators in OpenCV: `cv::GaussianBlur()` and `cv::addWeighted()`. Since the Gaussian blur executes a separable filter, this results in three OpenCL kernels being executed. By contrast, both Halide and our DSL are able to reduce this to two kernels. This results improved performance for both Halide and our DSL when compared to OpenCV, with Halide demonstrating a further small advantage in kernel execution time over our DSL. In the previous section, our benchmark for the Gaussian blur showed a 5 ms advantage for our DSL over OpenCV, which is consistent with the results shown here. Consequently, we must also assume that the performance difference between OpenCV and our DSL for the RGB unsharp mask is dominated by this difference.

Adding a conversion to and from the YCrCb colour space introduces an additional two kernels on the OpenCV implementation, along with a corresponding set of round-trips to global memory. This adds an additional 14 ms of kernel execution time. In our DSL implementation this colour conversion introduces no additional memory accesses and appears not to have a measurable performance impact. The Halide implementation currently appears to schedule poorly, greatly increasing kernel execution time. This may be resolvable through optimization and more intelligent scheduling, and warrants further investigation.

## 4.5 LIMITATIONS AND FUTURE WORK

Our DSL has a number of strengths. It provides a strongly typed abstraction which closely mirrors the textbook form of image processing operations, whilst transparently enabling kernel fusion and acceleration on SYCL. However, it also has a number of limitations.

The DSL described in this chapter only supports a subset of image processing operators. We have demonstrated support for point and local operators. Generalizing to other problem domains, map and stencil operations are well supported by this model. These operators are simple to schedule, typically requiring an expression to be evaluated for each pixel in an output image. Some geometric transformations such as rotations, warps and scaling are similarly supported. However, this leaves an important class of operators unsupported. Global operators such as histograms and Fast Fourier Transforms (FFTs), where the output is dependent on the entire in-

put image, do not fit well with our model. Many of these operators are supported by OpenCV and are implementable in Halide, making this a key weakness of our approach.

Despite this, we believe that this approach is sufficiently expressive to encompass more complex algorithms and that more work is warranted in exploring other computational patterns, such as reductions. This requires further work on the scheduling of work-items with our DSL library.

A further weakness of our approach relates to the static nature of metaprogramming in C++ and the compile-time nature of our implementation. Halide's primary strength lies not simply in the ability to generate efficient kernels through fusion and the elimination of intermediate memory operations. Instead, its strength is in the capability to rapidly explore possible optimizations through manipulation of how the evaluation of expressions in the language are scheduled. For example, variants of an algorithm can be optimized for a multi-core CPU by parallelizing across cores and vectorizing to generate SIMD instructions. Alternatively, a schedule suitable for a GPU can be generated by mapping pixels to work-items and exploiting tiling. This can be accomplished within Halide with only a few additional function calls, which makes iterating on potential optimizations extremely rapid. This is very challenging to replicate when using offline compilation and metaprogramming, and leads to extremely complex template code that is difficult to maintain.

Related to the question of scheduling, our approach currently lacks a cost model to guide fusion. A more automated model, such as that used by Fousek, Filipovic and Madzin (2011) appears attractive. However, such a model would have to be invoked during the template instantiation stage of compilation. Whilst it seems plausible that such a model could be integrated into our DSL through the use of C++11's constant expressions, this would add considerable complexity.

Whilst the capability to combine elementary functions into fused pipeline stages is valuable, and leads to measurable performance improvements, not all pipelines can be conveniently expressed within the syntax of our DSL. One fundamental limitation here is that expressions within the syntax of our language form trees. However, a number of image processing algorithms can be better represented as a more general directed graph. Whilst it is possible to implement a compile-time graph using C++ templates, doing so in a syntactically attractive and human readable form is challenging. Being able to express these more complex pipelines in a syntactically attractive manner within our DSL would be a desirable addition.

The dynamic generation of kernels at runtime also allows for a class of optimizations which cannot be easily replicated with offline compilation. When kernels are generated at runtime, operands which are uniform across work-items and for which their value is known at the point of kernel generation can be emitted into the kernel source code as constant literals, further reducing memory bandwidth and enabling compiler optimizations such as dead code elimination of branches which can then be deduced as inaccessible. Using PACXX (Haidl and Gorlatch, 2014), Haidl, Steuwer et al. (2016) have demonstrated a technique for multi-stage programming within GPU-accelerated C++ for which alleviates this problem. Given the strong similarities in approach between the ComputeCpp SYCL implementation (Codeplay Software Ltd., 2016) and PACXX, this may provide a route forwards on this particular limitation.

One further avenue for research would be to combine our DSL with our C++ compiler and programming model for HSA. Whilst kernel fusion is able to eliminate the cost of intermediate memory accesses by combining elemental operations into a single kernel in many cases, data movement remains a significant cost when using our DSL. In many use cases, transferring data in and out of SYCL's buffers exceeds the cost of computation. These data movement costs could potentially be eliminated by combining our DSL implementation with our HSA compiler and runtime, to be described in chapter 5.

## 4.6 DISCUSSION

In this chapter, we have described the implementation of a deeply embedded domain-specific language for image processing which utilizes SYCL to provide hardware acceleration on hardware supporting OpenCL 1.2.

In motivating the work described in this chapter, we identified three primary goals for our work on implementing DSLs on SYCL.

- To illustrate how high-level languages such as C++ can provide a route to managing the complexity of heterogeneous systems.
- To validate the suitability of SYCL as a foundation for building complex higher C++ libraries.

- To demonstrate that SYCL can be utilized to generate efficient kernels through the composition of simpler primitives.

This work has served both as a validation of features of the SYCL specification and as an evaluation of the performance of Codeplay's ComputeCpp SYCL implementation. One of the goals of SYCL is to provide a foundation which can be used to efficiently implement higher-level C++ programming models on OpenCL devices. This work, conducted before any SYCL implementations were made publicly available, represents an early example of such a use. By successfully implementing our DSL on SYCL, this work acts as a partial demonstration that SYCL is functionally sound, and contributes to our goal of providing validation of the SYCL specification.

We have demonstrated how, through the use of template metaprogramming, we can embed a small DSL within standard C++, and transform expressions written in that DSL into OpenCL kernels using only the SYCL compiler. This DSL enables us to provide convenient, familiar syntax to image processing domain experts, whilst abstracting away the complexities of OpenCL. This relates back to our core theme of providing performant domain-specific abstractions to non-machine experts.

Previous similar works such as Halide (Ragan-Kelley et al., 2013) and ArrayFire (Malcolm et al., 2012) have composed kernels through run-time manipulation of OpenCL C strings. Due to SYCL's offline compilation model, dynamic run-time construction of kernels within SYCL can only be achieved through the use of SYCL's OpenCL interoperability features. This approach requires the library developer to both implement the compilation machinery necessary to dynamically generate OpenCL C strings and to deal with translation issues between C++ and OpenCL C. We demonstrate that OpenCL kernels can be generated from a DSL statically at compile-time. By leveraging the shared-source nature of SYCL, it is possible to deeply embed a DSL which maintains the simple programming model of parallel primitive libraries, retains the strong type system of C++ and generates kernels at compile-time.

Due to SYCL's relative immaturity, little work has been done on evaluating SYCL's performance. We have demonstrated that the SYCL runtime and scheduler do not impose undue overheads, achieving equivalent host overhead to OpenCV across all of our benchmarks. We have also demonstrated that despite the comparative complexity of the templated expressions generated by our DSL, the ComputeCpp SYCL compiler is able to generate kernels with equivalent performance in many cases. Whilst our benchmarks do reveal some cases where the OpenCL kernels in OpenCV are able to out perform those generated by our DSL, this is due to algorithmic differences. Even

this case, we are able to replicate equivalent performance in SYCL by duplicating the implementation approach adopted by OpenCV. This provides validation that in addition to SYCL being functionally sound, the use of a high-level C++ abstractions does not impose unreasonable performance overheads.

Image processing pipelines composed of single operators are not the primary use case for our DSL. Instead, our focus is on combining primitives in order to generate efficient kernels. Kernel fusion can offer significant performance benefits in memory-bound applications such as image processing. We have demonstrated that in cases where an expression is composed of several operators, such as the desaturation and unsharp mask benchmarks, the combination of SYCL and expression trees can be utilized to generate fused kernels. This eliminates intermediate memory accesses, resulting in reduced bandwidth requirements and improved performance. This is not well addressed by libraries such as OpenCV, leading to applications which generate unnecessary off-chip bandwidth.

SYCL's model of implicit management of scheduling and data movement does not appear to impose undue performance overheads. However, it does introduce some trade-offs in terms of programming model. SYCL's buffer and accessor model imposes lifetime and scoping constraints on accessors. Within the implementation of our DSL, this necessitated the implementation of compile-time transformations between a host representation of our DSL, and a device-specific representation based on accessors. This transformation removes types which are not valid within a kernel such as SYCL's image and buffer types, and replaces them with accessors. This transformation would not be necessary when implementing our DSL using either CUDA or our HSA programming model described in chapter 5. Whilst this system of buffers and accessors is somewhat necessitated by the underlying OpenCL implementation, and it does succeed in simplifying the management of data movement, it also causes significant additional implementation complexity within our DSL implementation.

The approach used in this chapter has subsequently proved applicable to a number of related projects within Codeplay Software. The same issues solved in this work were also utilized in order to provide a SYCL backend to TensorFlow (Abadi et al., 2016) and the Eigen linear algebra library (Guennebaud, Jacob et al., 2010); and in the implementation of SYCL-BLAS (Aliaga, Reyes and Goli, 2017a,b), a Basic Linear Algebra Subprograms (BLAS) library for SYCL. Additionally, they provide the foundation of VisionCpp (Goli, 2016), a library providing accelerated computer vision operations on SYCL. These alternative application domains all share a common property with

our image processing use case, in that they all commonly feature combinations of element-wise maps and reduction operations. Mapping these operations in combination with expression tree-based kernel fusion to OpenCL's execution and memory models is relatively straight-forwards in comparison to domains which operations which require global communication such as fast-fourier transforms. Further work by Iwanski and Goli on implementing SYCL support in TensorFlow and Eigen (Goli, Iwanski and Richards, 2017) has revealed that the limitations of the buffer and accessor model raised in this chapter prove to be even more challenging in existing codebases which were not designed with SYCL's accessor model in mind.

The design of SYCL's system of accessors, and the lifetime constraints that come with it are rooted in the memory model of OpenCL 1.2, where the host and device address spaces may be disjoint. More modern heterogeneous architectures and runtime APIs, such as Heterogeneous System Architecture, have introduced support for shared virtual memory which greatly simplifies the sharing of data between PUs. In the next part of this thesis, we will discuss an alternative approach to C++ on heterogeneous systems, exploring a C++ compiler and runtime for Heterogeneous System Architecture which greatly simplifies the programming model.

# Part III

# Heterogeneous System Architecture

# 5 OFFLOAD FOR HETEROGENEOUS SYSTEM ARCHITECTURE

This thesis deals with the use of new standards and C++ programming models to ease the use of heterogeneous systems, applied to problems selected from the domain of visual computing. In the preceding chapter, we explored an approach to building a domain-specific language on top of SYCL. SYCL adopts a relatively high-level of abstraction, providing a C++ template library layered over OpenCL.

The subsequent two chapters (chapters 5 and 6) described a pair of closely interrelated projects, which were developed concurrently. Heterogeneous System Architecture (HSA) aims to provide a low-overhead toolkit aimed at enabling the development of parallel programming models and languages for heterogeneous systems. HSA provides a much lower level of abstraction than either SYCL or OpenCL, aiming to provide a base toolkit for the designers of new parallel programming models and languages, rather than exposing a model designed for application developers. This is coupled with more capable baseline hardware, providing pervasive support for Shared Virtual Memory (SVM), removing the need for SYCL's complex scheduler and intrusive buffer and accessor-based design.

In this chapter we describe the design and implementation of a shared-source C++14-based programming model for HSA. This work will later be utilized by RTKit (chapter 6), our framework for exploring ray tracing performance on HSA. With this programming model, we provide an environment where existing code can be easily utilized on heterogeneous devices with little or no modification to the source code. Additionally, due to HSA's unified virtual address space, we are able to share data between agents without the need for explicit memory copies or special container types. This leads to reduced latency and bandwidth overheads, and to improved performance in several benchmarks.

In chapter 4, we used SYCL as the foundation for implementing a Domain Specific Language (DSL) for image processing. The design of SYCL, and of the OpenCL 1.2 Application Programming Interface (API) underlying it, introduced some constraints on developers. The address space of memory buffers in OpenCL 1.2 is logically disjoint from that of host memory, and data movement requires explicit management.

SYCL attempts to ease this burden by using a complex scheduler and system of accessors to provide implicit data movement. Unfortunately, the use of specialized buffer types rather than pointers, and the restricted lifetime of accessors, can prove challenging to integrate into existing code bases. This also restricts SYCL 1.2 to providing coarse-grained coherency, preventing multiple accelerator devices from simultaneously manipulating the same region of memory.

Much of the complexity of both implementing and using SYCL is reduced by building upon HSA. Our model does not require a complex scheduler to track dependencies, and allows for the use of standard C++ pointers rather than opaque buffer types. However, this comes at the cost of increased hardware requirements.

Implementing our model required significant extension and modification of an existing compiler backend, followed by integration with Codeplay Software's Clang-based Offload (P. Cooper et al., 2010; Donaldson et al., 2010) compiler frontend. This compiler frontend was extended with additional functionality to support an HSA-based programming model. This enables the mapping of the complex segmented memory model found in HSA to C++ with fewer language extensions than existing models such as CUDA (Chakrabarti et al., 2012).

Finally, a C++14-based runtime library was produced to enable developers to develop applications using the compiler. This library provides the necessary APIs to allow developers to manage communication, synchronization and scheduling of work between multiple Processing Units (PUs) within a heterogeneous system.

The RTKit ray tracing framework described in chapter 6 utilizes this compiler and runtime library to provide acceleration on HSA-based devices, providing further verification of our approach.

We will begin the chapter by discussing the background and motivation for our work in section 5.1. This is followed in section 5.2 by a discussion of the design and features of our programming model.

Our model required both the extension of a compiler and the implementation of a supporting runtime library. Section 5.3 provides a high-level overview of the compilation model used by our compiler and runtime, while section 5.4 provides further practical implementation details on the compiler and section 5.5 provides details on the runtime library.

Several other authors have produced C++-based programming models for heterogeneous accelerators. We provide a comparison to CUDA, HCC, and SYCL in section 5.6. We also provide an evaluation of our compiler and runtime in section 5.7.

Finally, we end with a discussion of limitations and future work in section 5.8 and some concluding remarks in section 5.9.

## 5.1 MOTIVATION

HSA offers a number of features which might reasonably be expected to impact the type of workloads that can be offloaded to accelerators and the manner in which they are implemented. These include: reduced latency for kernel dispatches; the reduction or elimination of host-device memory copies; and system-wide communication through atomic operations and signals.

These are attractive properties for a compute runtime on an embedded system. Low-latency, low-power compute has a number of important applications, such as image processing in smartphones and computer vision in drones or autonomous vehicles.

The HSA specifications define a small C-based runtime API (HSA Foundation, 2015c). This API enables system introspection, and the creation, destruction and management of memory allocations, queues and signals. The HSA specifications additionally define the intermediate language, Heterogeneous System Architecture Intermediate Language (HSAIL) (HSA Foundation, 2015b). Further optional API extensions provide support for images and for finalizing HSAIL code into the native Instruction Set Architecture (ISA) of agents.

HSAIL and the HSA runtime API are sufficient to enable the development of small applications. However, developing software in HSAIL is comparable in complexity to working in an assembly language. This rapidly becomes an unproductive approach as applications scale in size. In order to make developing larger applications a more tractable option, a compiler and programming model for a higher-level language are required.

The work described in this chapter and the ray tracing work described in chapter 6 was begun prior to the publication of the HSA specifications. At that time, there were no high-level language implementations targeting HSA, and no HSA compatible hardware was publicly available. Indeed, the work described in these chapters

has helped inform Codeplay's input to the specifications throughout their development. In addition to our own work, this lack of compilers is now beginning to be addressed by works such as Numba (Lam, Pitrou and Seibert, 2015), Heterogeneous Compute Compiler (HCC) (Sander et al., 2015) and CL Offline Compiler (CLOC) (Rodgers, 2015). These compilers remain active research endeavours.

This lack of both runtime and hardware implementations directly motivated many of the design decisions taken in producing our model. As such, the design of the model aims to satisfy a number of goals:

- Enable validation of the HSA specifications
- Directly expose new functionality provide by the HSA platform
- Simplify experimentation and optimization

The goal of validating specifications is shared with our work implementing DSLs on SYCL, described in chapter 4. HSA is a new series of specifications and builds upon new hardware. Much like SYCL, HSA was envisioned as a foundation on which other high-level programming models, languages and runtimes could be developed in order to exploit the performance of heterogeneous systems. This work is one of the earliest examples of such a programming model and as such, provided valuable practical experience of working with early implementations.

Our model aims to provide a low-overhead interface allowing access to every feature of the underlying HSA implementation from C++. During the early stages of this work, the final performance characteristics of both the HSA compatible hardware and runtime library were unknown. Adopting a low level of abstraction simplifies measurement and exploration of those characteristics.

Given this uncertainty over the performance characteristics of available hardware, it is desirable to simplify moving code between host and agents in order to accelerate experimentation. Consider the example of transforming a simple Central Processing Unit (CPU) loop into a Graphics Processing Unit (GPU) kernel, with the ultimate goal of improving application performance. When using a dual-source model such as OpenCL, source code may need to be rewritten in the kernel language. This burden is reduced by single-source models such as CUDA, OpenMP or C++ AMP due to sharing a common language between host and accelerator code. However, these models still require the addition of compiler directives, keywords, special container classes and memory transfer operations. On a platform where the relative performance of heterogeneous processors is unknown, such experimental optimizations may

fail to produce the desired performance benefits and so developer effort is wasted. Where the burden of porting is high, this has the potential to discourage further experimentation. By minimizing the changes required to transform code into a form suitable for execution on a heterogeneous device, we can ease this burden and encourage exploration of potential optimizations.

Rather than attempting to bring a high-level framework such as SYCL to HSA, we chose to focus on a low-overhead, low-latency runtime and programming model, whilst seeking to retain the benefits of a comparatively high-level programming language. A high-level scheduling framework similar to SYCL could reasonably be built on HSA today. Both the SYCL specification and the ComputeCpp SYCL implementation were undergoing rapid evolution concurrent with this work.

We can further motivate our approach by means of an example. Listing 5-1 demonstrates the use of a shared ring buffer, accessed concurrently from the host processor and a kernel agent. The ring buffer consists of a single implementation which generates valid code for both the host processor and HSA kernel agents, and can be accessed concurrently by the host processor and multiple kernel agents simultaneously.

Such an example cannot currently be implemented in CUDA or C++ AMP due to the limitations of host-device synchronization under these models. Such an example could be implemented through the use of shared virtual memory in OpenCL 2.0. However, this would require separate implementations of the ring buffer class in OpenCL C and the host application language. Furthermore, such an implementation would require support for fine-grained system SVM, and at the time of writing no published OpenCL implementation supports this level of shared virtual memory.

Similarly, such an example cannot be implemented in SYCL 1.2 due to coherency model constraints. The SYCL 2.2 provisional specification introduces support for shared virtual memory. However, at the time of writing there are no complete implementations of SYCL 2.2 and any future implementation would also require an underlying OpenCL implementation with support for fine-grained system SVM.

In the subsequent sections we will describe the programming model, compiler and runtime used to implement this example.

```
// Declare a multi-producer, multi-consumer ring buffer on the host
// processor.
mpmc_ring_buffer<float> buf;

// Declare a C++11 atomic, which will be shared by both the host
// processor and the kernel agent.
std::atomic<bool> run = true;

// Start the kernel on a throughput processor (GPU, DSP) to
// continually dequeue items from the buffer and process.
auto future = rt::parallel_for<class dequeue>(rt::throughput,
                                              SIZE, [&](){
  // Poll the std::atomic for termination status.
  while (run) {
    // If the queue is not empty, dequeue an item from the shared
    // ring buffer and process.
    float entry;
    if (buf.try_dequeue(entry))
      ...;
  }
});

// On host CPU: loop continually, pushing items into the buffer.
while (run) {
  float entry = ...;
  buf.enqueue(entry);
}

// Wait for kernel completion.
future.wait();
```

**Listing 5-1**: Shared Ring Buffer Concurrently Accessed by Host Processor and HSA Kernel Agent

## 5.2 PROGRAMMING MODEL

In this section we will describe our programming model and its capabilities.

OpenCL defines the OpenCL C kernel language. This is based on C99 (ISO/IEC, 1999) but requires specific extensions and restrictions. Typically this either requires separate source files for host and device code, or the embedding of device code within strings in the host code.

By contrast, Offload enables the use of a single common language on both host and accelerator devices. This greatly simplifies the reuse of source code, and reduces the barrier to experimentation and investigation presented by needing to manually transform source code between languages when migrating code from host to device or vice-versa.

This single-source model is similar to that adopted by existing C++-based programming models for GPUs and other accelerators such as CUDA, SYCL and C++ AMP. However, the unified virtual memory system in HSA allows our approach to relax some of the constraints that we find in models intended for use with discrete accelerators where the address spaces used by host and accelerator processors may be physically and logically disjoint.

Most notably, we are able to pass data structures between agents by address, rather than relying on container types such as OpenCL's buffer or C++ AMP's `array_view` constructs. Not only does this result in bandwidth savings, it also greatly simplifies concurrent access to data structures from multiple agents by eliminating the need to keep multiple distinct copies of a data structure coherent.

Additionally, because addresses in HSA's global segment will remain valid and consistent across agents, we can make use of data structures that contain pointers as member variables. This enables the implementation of important data structures such as trees and linked lists without the need to make intrusive changes to accommodate API-specific container types.

### 5.2.1 Representing Kernels

In HSAIL, the declarations and definitions of kernel functions must be annotated with the `kernel` keyword. This serves as a guide to the finalizer, marking entry

points around which hardware-specific setup or scheduling code may need to be generated.

Consequently, our compiler and programming model must also differentiate kernel entry points from general functions. A kernel under our model is represented by a void-returning function, annotated with a generalized attribute, `[[hsa::kernel]]`. It may seem attractive to infer whether or not a function is a kernel entry point from it's usage and so avoid requiring the use of a generalized attribute. However, this approach would require more complex whole program analysis.

Listing 5-2 demonstrates a minimal vector addition kernel under our model. The unqualified pointer arguments are implicitly treated as addresses within the global segment. This example is equivalent to the OpenCL implementation shown in listing 2-6 and the HSAIL implementation in listing 2-7.

```
1 [[hsa::kernel]]
2 void vector_add(float *a, float *b, float *c) {
3   uint32_t i = rt::builtin::workitemabsid(0);
4   a[i] = b[i] + c[i];
5 }
```

**Listing 5-2:** A Vector Addition Kernel in our C++ Programming Model for HSA

Kernels may also be represented as lambda functions, as illustrated in listing 5-3. This comparable to the SYCL vector addition kernel illustrated in listing 2-2, in that it encompasses both the definition of the kernel function and kernel dispatch in a single example.

```
1 float a[count];
2 float b[count];
3 float c[count];
4
5 rt::parallel_for<class vector_add>(rt::throughput, count, [&](){
6   uint32_t i = rt::builtin::workitemabsid(0);
7   a[i] = b[i] + c[i];
8 });
```

**Listing 5-3:** A Lambda-based Vector Addition Kernel in our C++ Programming Model for HSA

Listing 5-3 also serves to illustrate one of the key strengths of our model. The arrays `a`, `b` and `c` are allocated within the current stack frame on the host processor. The

addresses of these arrays are captured as members of a lambda function object. This function object is constructed by the host processor, but evaluated on a kernel agent, such as a GPU. The kernel agent is able to dereference the addresses of these arrays directly, avoiding the need for expensive memory transfer operations.

However, Listing 5-3 also exposes a limitation of the C++ Application Binary Interface (ABI) and shared-source compilation. In listing 5-3, a lambda expression is used to represent the kernel. Lambda expressions are instances of unnamed class types called closure types (ISO/IEC, 2014, section 5.1.2). In section 3.2.1, we previously described how C++ compilers are able to generate unique symbol names for functions, partially based on type information. On Linux-based systems, this name mangling is accomplished according to rules defined in the Itanium ABI. The Itanium ABI (CodeSourcery et al., 2004, section 5.1.7) leaves the name mangling of closure types unspecified in some contexts and requires the inclusion of a counter parameter based on the lexical ordering of closure type in other contexts. This decision to leave the mangling of unnamed types unspecified is predicated on the assumption that such types are not externally visible outside the translation unit in which they are declared.

This assumption is valid when compiling for a single target architecture, where only a single compiler invocation is used. However, the use of two compilers to compile the same source file breaks this assumption, as it becomes necessary to make an association between the lambda functions found in the host and device files. Furthermore, due to the presence of the preprocessor, it is impossible to guarantee that the lexical order in which lambda functions are parsed is consistent between the host and device compilers. This adds further complexity to correlating unnamed types between two compiler passes. Consequently a method of providing consistent naming of kernels based on lambda functions is required.

A related issue exists in SYCL. SYCL allows for the use of separate host and device compilers produced by different vendors. These compilers may use different mangling schemes in cases were it is not mandated by the ABI. SYCL requires a unique type parameter for the `parallel_for` template function, and uses this to link the host and device representations of the kernel. We adopt the same strategy in our programming model.

### 5.2.2 Kernel Dispatch

Kernel dispatch in HSA is accomplished by writing an Architected Queuing Language (AQL) packet into a ring buffer, as described in section 2.7.2 and illustrated in listing 2-5. Within our language runtime library, this dispatch process is encapsulated in the `parallel_for` member of the `queue` class. This method takes a pointer to a host function as an argument, and uses it as a key to locate a kernel for the associated agent. The details of this process are further described in section 5.5.

```
// Create a hardware-managed queue for processing AQL packets on the
// specified agent.
auto queue = agent->create_queue(4096);

// Configure the grid and work-group extents, dynamic group memory,
// memory fence behaviour etc.
rt::kernel_dispatch_info info;
info.range                 = rt::nd_range{{16384,1,1}, {256,1,1}};
info.dynamic_group_mem_size = 1024;
...

// Enqueue a kernel by providing the dispatch configuration, a
// function pointer to the kernel, and the kernel arguments.
float* a, b, c;
auto future = queue->parallel_for(info, vector_add, a, b, c);
future.wait();
```

**Listing 5-4:** Enqueuing a Kernel to a Specific Agent and Queue

Listing 5-4 demonstrates enqueuing a kernel function using our programming model. The `parallel_for` method enqueues a kernel to a specific queue, and consequently specific kernel agent. The arguments consist of the dispatch configuration such as the grid extents, a function pointer corresponding to the kernel function, and a variadic set of arguments to be passed as kernel arguments. The `parallel_for` function returns a completion future.

This object is an abstraction of an HSA signal, and provides methods for querying the execution status of the kernel, synchronizing execution, or triggering further computation on kernel completion. The future is returned immediately after a kernel has been enqueued, while kernel execution occurs asynchronously. Where synchronization is required, calling the `wait` member function on the returned completion future will cause execution to block until the kernel has completed execution.

For convenience, we also provide a second form of dispatch as a free function. This form accepts an execution policy as the first argument, allowing a user to indicate a preference for a latency or throughput-optimised agent. Based on this argument, the runtime will select a suitable agent and queue on which to dispatch work. This is illustrated in listing 5-5.

```
// Enqueue a kernel by providing an agent selection policy, the
// dispatch configuration, a function pointer to the kernel, and
// the kernel arguments.

// This example also demonstrates implicit construction of a
// kernel_dispatch_info argument from a work-item count.
float* a, b, c;
auto future = rt::parallel_for(rt::throughput,
                               16384,
                               vector_add, a, b, c);
future.wait();
```

**Listing 5-5:** Enqueuing a Kernel to an Agent Selected by the Runtime Library

We implement the dispatch process as a thread-safe multi-producer queue, allowing for multiple threads or agents to submit work to a single queue concurrently. A single agent may have many concurrently active queues, and a heterogeneous system may contain many agents. Each queue may process packets out of submission order, relative to any other queue within the system. Within a single queue, kernel dispatch packets will begin execution in submission order. However, even in this case kernels are allowed to begin execution as soon as all preceding work has launched, rather than completed.

Given this complexity, it is clearly necessary to be able to express dependencies between kernels and synchronize execution between agents. We provide several approaches for this.

AQL packets contain flags which may be set to introduce agent or system-wide memory fences or to require that all preceding packets submitted to the queue have completed execution. These properties can be controlled using the dispatch configuration parameter of the `parallel_for` function. This is typically sufficient to resolve dependencies between kernels dispatched to a single queue.

For resolving more complex dependencies, we can use futures and signals. These types may be used from both host and agent code and enable system-wide querying and waiting. Listing 5-6 illustrates one possible example of this. Two kernels are

expressed as lambda functions and enqueued to separate queues. The first kernel waits indefinitely for a signal to be set to a non-zero value. This kernel can be preempted and suspended until the signal condition is satisfied. The second kernel is dispatched to a different queue, and therefore also potentially a different agent. This kernel increments the signal, allowing the first kernel to continue.

```
// Create a new signal and set the initial state to zero.
rt::signal sig{0};

queue_a->parallel_for<class wait_on_signal>(grid_size, [&]() {
  // Enter a low-power state and wait indefinitely for signal to be
  // set to a value >= 1. Signals can wake spuriously, so we need to
  // verify the condition.
  uint64_t value = 0;
  do {
    value = sig.wait(gte, 1, UINT64_MAX, wait_blocked);
  } while (value < 1)
});

queue_b->parallel_for<class increment_signal>(grid_size, [&]() {
  // Increment the signal.
  sig.add(1, order::scar)
});
```

**Listing 5-6:** Using Signals for Communication and Synchronization

This approach allows for multiple active kernels to communicate. However, it requires full profile support. This is because full profile implementations are required to guarantee independent forward progress across multiple queues, while base profile agents are not (HSA Foundation, 2016a, p. 31). For further discussion of the differences between full and base profile, refer to section 2.7.3. Without a guarantee that queues will make forward progress independently, a kernel waiting for a condition to be satisfied may deadlock process execution by preventing the execution of a second kernel which would ultimately satisfy the wait condition and allow forward progress.

Alternatively, we can utilize AQL barrier packets to express compound dependencies. These packets express a dependency on multiple signals. These dependencies may be resolved by the packet processor monitoring the queue on which they are dispatched without the intervention of the host CPU.

Listing 5-7 illustrates the usage of barrier packets. Here two kernels are dispatched to different queues, and potentially separate agents. This is followed by a barrier

```
// Dispatch a kernel to queue A.
auto future_a = queue_a->parallel_for(...);

// Dispatch a second kernel to queue B, potentially on different agent.
auto future_b = queue_b->parallel_for(...);

// Dispatch a barrier packet to ensure both kernels have completed.
auto future_c = queue_b->barrier_and(future_a, future_b);

// Wait for completion.
future_c.wait();
```

**Listing 5-7**: Using Barrier Packets to Express Kernel Dependency Graphs

packet which will stall execution on the queue on which it is dispatched until all dependencies are satisfied. This tracking of dependencies may be efficiently managed by a hardware packet processor without the need for intervention from the host agent.

### 5.2.3 Function Annotations

In general, we do not require functions intended for execution on kernel agents to be annotated differently from functions intended to be executed on the host processor, i.e. there is no need for the `__device__` or **`restrict`**`(amp)` annotations that appear in CUDA and C++ AMP. Instead, we only require that kernel functions are annotated with a generalized attribute, `[[hsa::kernel]]`, and that a second attribute, `[[hsa::function]]`, is applied to externally visible functions that will be required for linking with HSA code generated from another translation unit. This enables us to identify the roots of the call-graph for kernel code. By traversing the call-graph of each kernel and annotated externally visible function, we are able to extract the full set of functions requiring compilation to HSAIL. This represents a generalization of work previously described by Donaldson et al. (2010) and P. Cooper et al. (2010) to HSA's more complex memory hierarchy.

We illustrate this with an example in listing 5-8. We require that the kernel function, `g`, is annotated. However, we do not require further annotations on the function `f`, and `f` may be called from both host and device code. In CUDA and C++ AMP, this function would have to be annotated with `__host__ __device__` or **`restrict`**`(amp,cpu)` respectively. Whilst adding these annotations in CUDA or C++ AMP is a simple

```
// This function requires the __device__ annotation in CUDA,
// restrict(amp) in C++ AMP, but no modification under our model.
void f() { }

// Kernel functions must be marked with the [[hsa::kernel]]
// generalized attribute under our model,
[[hsa::kernel]] void g() {
  f();
}
```

**Listing 5-8:** Kernel and Function Annotations

task, their impact is viral. Each function called from an annotated function must also be annotated. In large, complex C++ applications this can often mean that the decision to use an existing class or function within a kernel can necessitate widespread modifications throughout the source code.

### 5.2.4 Memory Segments

As discussed in section 2.7.4, HSA subdivides its virtual address space into a number of segments. These segments are encoded in the representation of all instructions that operate on memory, and consequently must be handled by the compiler.

For simplicity, it might appear attractive to avoid the complexity of a segmented address space, and build a programming model which operates entirely on flat addresses. This approach would have a number of undesirable consequences.

Under HSA's large machine model, flat addresses are defined as 64-bits, while private and group addresses are 32-bits. Consequently pointer arithmetic on flat addresses must also be performed using 64-bit arithmetic. This may lead to increased register pressure on architectures such as Advanced Micro Devices (AMD)'s Graphics Core Next (GCN) (Advanced Micro Devices, 2016a) where 64-bit values are stored using a pair of consecutive 32-bit registers. This can lead to increased register spilling, or reduced kernel occupancy.

The flat segment does not encompass the read-only, kernarg, arg and spill segments. Several of these segments (kernarg, arg and spill) would provide little additional functionality by exposing them directly to developers and can be handled entirely transparently by the compiler. However, the read-only segment offers potential performance benefits and it would be undesirable to exclude its use. A similar issue

exists in OpenCL, where the *generic* address space does not encompass *constant* addresses.

A possible naive approach to implementing support for flat addresses in hardware is to issue requests to multiple memory units simultaneously, and for those units to reject requests which fall outside their supported address range (HSA Foundation, 2016b, p. 315). On such hardware, the use of flat addresses will increase pressure on the memory units by limiting the opportunity to service requests to multiple segments in parallel.

All segments in HSA are logically disjoint. Consequently, addresses can be assumed not to alias if they lack a common segment. This information aids alias analysis in the compiler and finalizer, and may lead to performance improvements. More generally, including explicit segment information in the generated HSAIL will provide the finalizer with more information on which to base optimization decisions.

The combination of these factors led us to conclude that our model should be capable of tracking segment information, and not simply rely on HSA's flat addressing support.

Three of HSA's segments can be handled entirely by the compiler and do not need to be exposed to users of our programming model. In HSAIL, the kernarg and arg segments must be used to pass kernel and function arguments respectively. Under our programming model, kernel arguments are implicitly members of the kernarg segment, and do not require explicit qualification with segment attributes. Similarly, the arguments of functions called within a kernel dispatch are implicitly members of the arg segment and do not require explicit qualification. The spill segment is also not exposed to users in our programming model. Whilst the compiler backend may spill allocations into this segment where necessary, this is handled transparently within the compiler. Consequently, generalized attributes are not provided for the spill, arg and kernarg segments.

The remaining segments are handled through a two-pronged approach. We extend the C++ type system such that every type is augmented with segment information. These segments act as type qualifiers, similar to cv-qualifiers in C++ (ISO/IEC, 2014, p. 74). The segments interact with the C++ type system in same manner, allowing for function overloading and template specialization by segment. As a type qualifier, this segment information will propagate through both type-based expressions such as templates, and through pointer arithmetic. We make use of C++11's generalized attributes to enable this extension of the type system, and define a set of rules re-

garding the use of these annotations along with the cases in which our programming model will implicitly infer the presence of these qualifiers without their explicit use. The use of these annotations is described in section 5.2.5, while the mechanism by which this implicit inference is implemented is described in section 5.2.6.

This approach differs from the model adopted by CUDA and HCC. These works define a programming model where explicit address spaces are only defined on variable definitions, and are not carried along with addresses in the language. A type inference algorithm such as Hindley-Milner (Damas and Milner, 1982; Hindley, 1969; Milner, 1978) can then be applied later to reconstruct this information in the compiler backend. This implementation approach is used by CUDA (Chakrabarti et al., 2012), HCC (Sander et al., 2015) and GPUCC (Wu et al., 2016).

### 5.2.5 Segment Inference

In the preceding section, we described address segment state as integrated into the type system, accessible through a set of C++11's generalized attributes. Mandating the use of explicit segment annotations on all variables would both place an unnecessary burden on developers and lead to incompatibility with existing C++ code which is designed without such a model in mind. Instead we define a set of rules for the inference of segment information where it is not explicitly provided by the application developer. In this way compatibility with existing C++ code can be preserved while retaining the control to override the default behaviour.

Adopting an entirely implicit model presents its own challenges. Whilst private segment variables can be easily identified by scope, group, global and read-only segment variables cannot be easily differentiated without some form of additional annotation.

We will begin by discussing the rules for the declaration of variables under our programming model, and the cases in which the explicit use of the segment-qualifier attributes are required.

Variable declarations in the group and read-only segments always require explicit segment qualifiers. This is because there is insufficient information to automatically disambiguate global variables allocated in the global, read-only and group segments by context alone. These variables may be declared as either program-scope global variables, or as static storage duration variables. Unlike OpenCL, we allow the de-

claration of group segment variables in any function within the call-graph of a kernel function, rather than restricting declarations to kernel function scope. Variables in the group segment are uninitialized by default, and constructors for class variables and arrays thereof will not be called. Where the calling of constructors is required, it can be accomplished through the use of placement `new`.

Segment-unqualified program-scope global and static storage duration variables are members of the generic address space and implicitly treated as members of the global segment.

Segment-unqualified variables with automatic storage duration declared within the lexical scope of a kernel function, or any function within the call-graph of a kernel function, are implicitly treated as allocations in the private segment. By contrast, segment-unqualified automatic variables in host code are treated as members of the generic address space, and consequently the global segment.

We illustrate this implicit behaviour with an example in listing 5-9.

```
1  // A function called from both host and kernel code.
2  void f() {
3    // An automatic storage duration variable implicitly allocated
4    // in the private or global segment, depending on the call site
5    // of f().
6    int a;
7  }
8
9  // A kernel function.
10 [[hsa::kernel]] void k() {
11   // An automatic storage duration variable implicitly allocated
12   // in the private segment.
13   int b;
14   f();
15 }
16
17 // A host function.
18 void h() {
19   // An automatic storage duration variable implicitly allocated
20   // in the global segment.
21   int c;
22   f();
23 }
```

**Listing 5-9:** Segment Inference for Automatic Storage Duration Variables

A variable `b` is declared within the lexical scope of kernel `k` and is treated as being an implicit member of the private segment. Another variable, `c`, is declared in a function which is only executed on the host processor. Consequently, this variable will be allocated in the stack frame on the host processor, and is implicitly a member of the global segment. Finally, variable `a` is declared in a function which is used in both contexts. Two copies of this function will be generated, and the segment in which `a` is allocated will dependent on the calling context. This duplication process is further described in section 5.2.6.

Non-static member variables may never be declared with an explicit segment qualifier. Instead, this is inferred from the parent object.

As described above, it is not necessary to annotate declarations of automatic storage duration variables. Despite this, a generalized attribute, `[[hsa::private]]` is also defined to accommodate cases where an explicit segment-qualified type is required for the purposes of disambiguation. A corresponding attribute for the global segment is similarly provided. One such example of this is function overloading. Listing 5-10 provides an example of this, showing the declaration of two different overloads of the function `f` which may be selected between based on the segment of the arguments to the function call.

```
1 void f([[hsa::private]] int *);
2 void f([[hsa::global]] int *);
```

**Listing 5-10:** Manually Overloading Functions by Segment

As shown in listing 5-10, pointer and reference types may also be qualified with a segment annotation. These qualifiers can be provided explicitly, or inferred from an initializer. We map all other segment-unqualified pointers and references to the global segment. This is motivated by the desire to share unannotated data structures containing pointers between the host and kernel agents.

HSA defines flat addresses as members of a virtual segment that acts as a superset of the private, group and global segments. As a virtual superset of the private, group and global segments, attempting to allocate a variable within such a virtual segment would result in the physical storage requirements of such a variable being underspecified, and so this is forbidden. Consistent with this, our programming model allows the use of flat segment annotation as type qualifier, including as a qualifier to pointer and reference types, but prohibits its use on variable declarations.

### 5.2.6 Call-graph Duplication

The memory segment on which load, store and atomic instructions operate is encoded as part of the instruction representation in HSAIL. Furthermore, under the large machine model, the size of addresses vary according to the segment with which an address is associated. A consequence of this is that several different implementations of a function may be required for any function with pointer or reference arguments. One approach to this is to require the programmer to manually define such alternative implementations. Such a requirement exists in OpenCL C 1.2, with the further caveat that C also lacks support function overloading. Our model provides support for this approach. A programmer may choose to explicitly overload a function based on the segment with which any pointer or reference argument is associated, as illustrated in listing 5-10.

However, this approach rapidly becomes onerous. Given a function with $N$ pointer or reference declarations in its parameters, and $M$ possible segments, an upper-bound of $M^N$ additional implementations of a function may be required. These pointer or reference parameters may be explicit, or implicit, e.g. the *this* pointer on non-static member functions, or the implicit pointer added to return structures. Furthermore, we cannot resolve this issue simply through the use of flat addresses throughout the application. Whilst the flat segment is defined as a superset of the private, group and global segments, it does not encompass the read-only segment, and so the issue persists[1].

We resolve these issues through the use of automatic call-graph duplication, as described by Cooper et al. (P. Cooper et al., 2010) and Donaldson et al. (Donaldson et al., 2010). These works describe an approach to compiling C++03 for the Cell Broadband Engine (BE) such as is found in the Sony PlayStation 3. This system had a number of properties that are comparable to modern heterogeneous systems. The Cell BE processor included a single Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs). The PPE used the Power Instruction Set Architecture v.2.03, whilst the SPEs utilize a specially developed and incompatible instruction set. Each SPE has its own local memory, but is unable to access the main system memory or the local memory of other SPEs directly, relying instead on Direct Memory Access (DMA) commands. In this context, call-graph duplication was used to generate functions in the two different instruction sets, and for two disjoint address spaces. We generalize this approach to support the full set of HSA's memory segments.

1 The HSA 1.0.3 runtime for AMD devices also lacks support for flat addresses on the private segment.

The call-graph duplication begins by identifying all kernel functions within a translation unit, based on the presence of the `[[hsa::kernel]]` attribute. We additionally identify any functions marked as available for external linkage within HSAIL, deduced from the `[[hsa::function]]` attribute. All of these functions are duplicated, and the function bodies reevaluated. During this reevaluation, the types of any automatic storage duration variables are modified to mark them as members of the private segment, and the types of pointer and reference variables automatically extended with segment qualifiers inferred from their initializers. This process only applies to variables that have not been explicitly annotated with a segment-qualifier. In other words, it will transform variables from the generic address space to a segment-specific address space, but will not remove a pre-existing segment qualifier. This process is a key component in allowing us to take unannotated standard C++ and transform it into a form compatible with HSA's segmented address spaces.

This process of type modification requires further semantic analysis to be performed on the duplicated function body, including reevaluating overload resolution and template instantiation. This process is applied recursively to all functions called from the duplicated function.

In cases where a function call cannot be resolved due solely to a mismatch between the segments of the parameters declared in the callee signature and the arguments of the call expression, the callee function is duplicated and its parameters are modified to correspond to the segments of the arguments of the triggering call expression. The body of this duplicated function is then reconstructed based on these updated argument as described above. This may result in the generation of multiple copies of a function which differ solely on the segment-qualifiers of their arguments.

Listing 5-11 provides an example of this duplication process. Here we define a function, `f`, with a single unqualified integer pointer parameter. Within the kernel function, `k`, we find two calls to `f`. The argument to the first function call is the address of a local variable within the scope of a kernel function (`&p`), and is implicitly a member of the private segment. The argument for the second function call (`&g`) is the address of a variable allocated in the group segment. Due to the segments of the pointer arguments, neither call expression matches the original declaration of `f`. Therefore, our compiler creates two duplicates of `f` and modifies the signature of each duplicate to match the arguments found in the call expression. It will then rebuild the body of each duplicate taking into account the modified signature to ensure correctness. Finally, the call expressions are updated to reference the duplicated functions.

```
// A function accepting a pointer argument.
void f(int *i) { *i = 0; }

// An integer allocated in the group segment.
[[hsa::group]] int g;

[[hsa::kernel]] void k() {
  // An integer implicitly allocated in the private
  // segment.
  int p;

  // Neither call matches the declaration:
  //   void f(int *)
  //
  // Our compiler creates automatically creates duplicate functions
  // based on the segment-augmented types of function arguments, and
  // then re-evaluates the function body based on the updated types.
  //
  //   void f(int[[hsa::private]] int *)
  //   void f(int[[hsa::group]] int *)
  //
  // This re-evaluation may require reapplying template instantiation
  // or overload resolution.
  f(&p);
  f(&g);
}
```

**Listing 5-11**: Automatic Call-graph Duplication

### 5.2.7 Sharing Data-Structures

Due to disjoint address spaces, the programming models adopted by OpenCL 1.2, SYCL and C++ AMP typically require that data required for processing on an accelerator be copied into some form of device-accessible buffer. We illustrate this process with an OpenCL 1.2 example in listing 5-12. In order to execute the OpenCL vector addition kernel, three OpenCL buffers must be created and initialized. The two input buffers are then copied to the OpenCL device. The arguments for the kernel are then specified and the kernel enqueued. Finally the result of the computation must be copied back to the host.

In contrast to this, the unified virtual address space in HSA allows us to reason that a valid CPU pointer is equivalent to a pointer in the global segment. This enables us to pass the host pointers directly as kernel arguments and dereference them on a kernel agent without the need for an intermediate copy. This allows us to eliminate the overhead of populating device buffers and copying them from host to device.

Furthermore, the use of call-graph duplication within our programming model allows us to dispense with function annotations such CUDA's `__device__` annotation. This combination of call-graph duplication and the mapping of segment-unqualified pointers to the global segment enables the sharing of unannotated standard C++ data structures between the host processor and kernel agents in a manner that is not possible under existing General Purpose Graphics Processing Unit (GPGPU) programming models.

Listing 5-13 illustrates this using a parallel map lookup as an example. Here, we define a kernel using a lambda function. However, we allow the lambda function to capture variables from the surrounding scope by reference, rather than by value. The lambda function captures a `std::unordered_map` object (`map`) and a `std::vector` (`output`) by reference. Each work-item then queries its unique index within the grid, and uses that index as a key to perform a lookup on the map and storing the result into the vector, `output`. We made *no modifications* to the implementations of `std::vector` or `std::unordered_map` to accomplish this.

The references to `map` and `output` within the lambda function correspond to addresses in the global segment. Our compiler will then create duplicates of any functions called directly or indirectly from the kernel, in this case the `find` member function of `std::unordered_map`, the subscript operator of `std::vector` and any further functions referenced by them.

```
// 3 vectors providing storage for the computation inputs/output.
std::vector<float> a, b, c;
...
size_t SIZE_IN_BYTES = sizeof(float) * count;

// Create 3 OpenCL buffers.
cl_mem a_buf = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                              SIZE_IN_BYTES, nullptr, nullptr);
cl_mem b_buf = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              SIZE_IN_BYTES, nullptr, nullptr);
cl_mem c_buf = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              SIZE_IN_BYTES, nullptr, nullptr);

// Enqueue copying of input data from std::vectors b/c to OpenCL
// buffers b_buf/c_buf.
clEnqueueWriteBuffer(queue, b_buf, CL_TRUE, 0, SIZE_IN_BYTES,
                     b.data(), 0, nullptr, nullptr);
clEnqueueWriteBuffer(queue, c_buf, CL_TRUE, 0, SIZE_IN_BYTES,
                     c.data(), 0, nullptr, nullptr);

// Set the buffers as kernel arguments.
clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_buf);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_buf);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_buf);

// Enqueue the kernel.
clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, &count, nullptr,
                       0, nullptr, nullptr);

// Enqueue copying of results back to std::vector a.
clEnqueueReadBuffer(queue, a_buf, CL_TRUE, 0, SIZE_IN_BYTES,
                    a.data(), 0, nullptr, nullptr);

// Synchronize execution.
clFinish(queue);
```

**Listing 5-12:** Host Code to Enqueue an OpenCL Kernel

```
#include <rt/compute.h>
#include <unordered_map>
#include <vector>

const size_t SIZE = 1024;

int main() {
  rt::initialize();

  // Declare an unordered map and populate it with data.
  std::unordered_map<uint32_t, float> map;
  populate_map(map);

  // Declare a vector to receive results.
  std::vector<float> output(SIZE);

  // Run a kernel to perform the map lookup in parallel.
  auto future = rt::parallel_for<class parallel_map>(rt::throughput,
    SIZE, [&]() {
    // For simplicity, we use the work-item ID as the search key.
    uint32_t i = rt::builtin::workitemabsid(0);

    // Perform lookup on map, captured by reference from outer scope.
    auto iter = map.find(i);
    output[i] = iter->second;
  });

  future.wait();

  // output is now populated with the results of the lookup.
  rt::terminate();
}
```

**Listing 5-13:** Unmodified Standard Template Library Classes Used Within a Kernel Lambda Function

This sharing of unmodified standard C++ classes between the host processor and kernel agents is one of the strengths of our approach. This is only possible due to the ability of our model to operate directly on pointers without container types and only minimal need for non-standard attributes, which can often be hidden by the API. Indeed, all of the user code in listing 5-13 is standard C++.

There are currently some limitations in what we can accomplish through this approach. For example, dynamic memory allocation using the default allocators will not function correctly. This precludes the use of functions such as `resize` on the vector or `insert` on the map. However, we can provide our own custom allocators, which utilize agent-dispatch packets to request memory allocation via the host processor. More generally, we do not currently support virtual function calls or exception handling.

## 5.3 COMPILATION MODEL

In this section we discuss the compilation model used by our compiler and runtime.

Much like SYCL, our compiler adopts a shared-source model, where code for execution on both the host processor and kernel agents may be contained in the same translation unit. However, the implementation approaches differ.

The SYCL device compiler included in ComputeCpp generates a header file with the kernel representations embedded. This file is automatically included when the host compiler parses the same source file, and is used to match kernel dispatches is the host code to device kernels.

For our HSA-based implementation, compilation is a two phase process consisting of offline and runtime stages.

During the offline phase, each translation unit that contains code intended to be executed on an HSA kernel agent is compiled twice. One compilation pass generates host code, emitting an Executable Linkable Format (ELF) object in the ISA of the host processor (typically x86_64). A second compilation pass selectively identifies and compiles the subset of functions that are required for execution on the kernel agents. Call-graph duplication is applied at this stage. These functions are compiled into Brigantine, the HSAIL binary format (BRIG), and the BRIG binary is embedded into

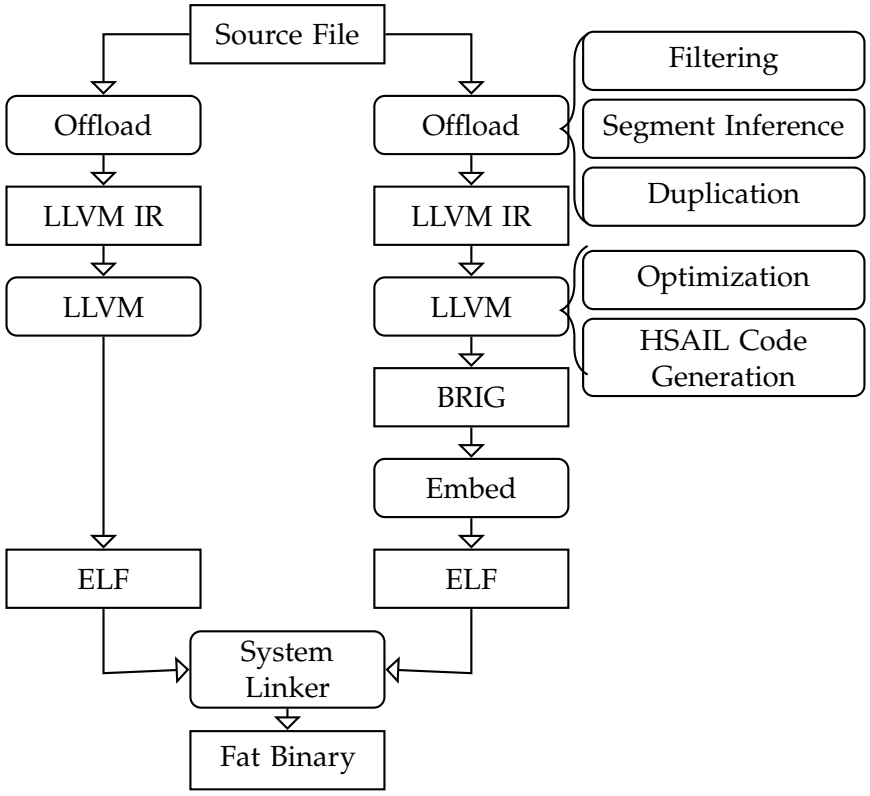



**Figure 5-1:** Compilation Flow for our C++ Programming Model

a specially named section of an ELF object. These object files can then be linked into a final executable using the standard ELF toolchain.

Unlike SYCL, we must require that the same compiler is used for both host and kernel compilation. This is due to the tighter integration of host and kernel code in our model. In order to ensure consistency of data structure layouts across compilers, SYCL places strict constraints on what data types are valid for use within kernels. This precludes the use of many common C++ features, such as static member variables, classes containing virtual functions and many forms of inheritance. We remove these constraints, allowing any valid C++ type to be referenced within a kernel.

Additionally, our HSA runtime also relies upon consistent name mangling between host and kernel compilation in order to link host and device representations of kernels and global variables. In initially describing name mangling in section 3.2.1, we noted that this process involves augmenting identifiers with additional type information for the purposes of disambiguation. In sections 5.2.4 and 5.4.2, we describe encoding memory segment information as type qualifiers, modelled after named address spaces (ISO/IEC, 2006, p. 37-39). The Itanium ABI (CodeSourcery et al., 2004)

does not provide support for named address spaces. Therefore, in order to support HSA's segmented address space, it was necessary to extend the name mangling scheme and we require that this extended scheme is supported by both host and kernel compilers. This extended scheme is designed to preserve binary compatibility with the existing Itanium ABI such that variables and functions that do not depend on segment-specific types will generate identical symbol names under both schemes. This enables us to link host code against existing C++ libraries, such as implementations of the C++ standard library, without modification.

The second phase of compilation is performed at runtime. HSAIL and BRIG are device-agnostic intermediate formats and not natively executable by the kernel agents in an HSA system. In order to execute a kernel, the kernel must be transformed from its intermediate form into the native instruction set of the agent on which it will run.

The BRIG objects are loaded from the executable at runtime, linked to resolve cross translation unit function calls and global variables, and then finalized into the native ISA of each kernel agent in the system. This is described in greater detail in section 5.5.7.

## 5.4 COMPILER IMPLEMENTATION

The programming model described in this chapter required the implementation of a compiler capable of generating HSAIL from C++. This compiler was implemented by integrating Offload, Codeplay Software's Clang-based compiler frontend, with an LLVM compiler backend provided by the HSA Foundation. Both the Offload frontend and LLVM backend were then further extended to support HSA functionality.

Offload extends the existing address space support within Clang to provide the call-graph duplication described in section 5.2.6. The call-graph duplication functionality in the compiler frontend was primarily implemented by Uwe Dolinsky and Victor Lomüller, whilst the HSA specific functionality; integration and extension of the LLVM backend represents my own work.

### 5.4.1 Integrating the HSAIL backend

The HSAIL LLVM backend initially received from the HSA Foundation was an early release to HSA Foundation members based on LLVM 3.2. As the official release of LLVM 3.2 dates from December 2012, this represented an outdated LLVM implementation and was incompatible with Offload and current releases of Clang. The use of LLVM 3.2 can be attributed to the HSAIL backend deriving from an OpenCL Standard Portable Intermediate Representation (SPIR) 1.2 backend.

The HSAIL backend received was capable of consuming simple OpenCL SPIR 1.2 modules and transforming them to HSAIL or BRIG. Many core OpenCL 1.2 features were functional. However, support for optional OpenCL features and many HSA features without equivalents in OpenCL 1.2 was missing, incomplete or untested. For example, support for function calls, vector operations, atomic instructions, signals and images were either missing or non-functional. Intrinsic functions were not directly accessible to a compiler frontend. Instead a bitcode file was provided that implemented the OpenCL 1.2 kernel built-in functions. The lack of higher-level language front ends meant that limited testing had been performed.

As such, significant work was required to update the HSAIL backend to a more modern LLVM release, implement missing or incomplete intrinsic functions and make them accessible to the compiler frontend and resolve code generation errors.

The HSAIL backend has since been updated and maintained to be compatible with LLVM 3.8. The backend is now capable of generating all HSAIL instructions, either through the LLVM instruction selection mechanism for common arithmetic instructions (i.e. add/mul etc.), or through intrinsic functions for special HSAIL instructions such as `workitemid`.

This updated LLVM backend was then integrated into Clang. Clang was extended with additional targets to allow targeting of both 32 and 64-bit HSAIL/BRIG using the HSAIL LLVM backend. These targets are enabled using the mechanism for selecting a cross-compilation target, i.e. `"-target hsail"` or `"-target hsail64"`.

Clang was further extended with target-specific preprocessor definitions and built-in functions, which ultimately map to HSAIL instructions. For example, the built-in function `__hsail_workitemid(uint32_t)` can be utilized from C or C++ when targeting HSAIL and will be lowered to a `workitemid_u32` instruction.

Invoking Clang with a HSAIL target triple will cause Clang to operate as a normal C or C++ compiler, without any further Offload extensions and emitting object code expressed in HSA's virtual instruction set as HSAIL or BRIG . HSA's virtual instruction set, HSAIL and BRIG are described further in section 2.7.5.

### 5.4.2 Segments and Address Spaces

In section 5.2.4, we described how every type in our programming model carries segment information. In this section we will discuss how this is implemented within our compiler.

Clang and LLVM have built in support for address spaces. This is modelled after the named address space (ISO/IEC, 2006, p. 37-39) support described in ISO/IEC TR 18037:2006, an ISO technical report on extensions to C99 for embedded processors. For the purposes of disambiguation, for the remainder of this section we use the term address space to refer to ISO/IEC TR 18037:2006 and to Clang and LLVM's address space support, and segment to refer to the specifics of HSA and our programming model.

We implement support for HSA's segments using Clang and LLVM's address space support. Clang and LLVM's support for address spaces have previously been used in this manner to provide support for other heterogeneous programming models, and GPU compiler backends. Clang provides OpenCL C and CUDA frontends which rely upon this address space functionality, while the Parallel Thread Execution (PTX) and AMDGPU LLVM backends are dependent on LLVM's address space support. Clang and LLVM represent address spaces as integral attributes, with zero representing the generic address space.

Mapping between the segment-qualifiers carried by types in our programming model and the address spaces used within Clang and LLVM introduces some challenges. These issues are rooted both in the lack of support for named address spaces in standard C++, and in design decisions made within Clang and LLVM.

The primary challenge relates to the mapping of the generic address space. Neither standard C++ nor the X86 backend used for host compilation support named address spaces. In other words, all addresses and allocations in host code map to the generic address space. Furthermore, all pre-existing standards-conformant C++ code, including the C++ standard template library, has been implemented under this

model. In order to retain compatibility with existing code, this property must be preserved.

From the point of view of a kernel agent, all host addresses are in the global segment, regardless of whether they represent a host stack or heap address. It is desirable to preserve this behaviour, as it allows for pointer-based structures such as lists to be utilized without being forced to explicitly annotate all of the pointers. This suggests that we should map the generic address to the global segment.

As we describe in section 5.2.5, this introduces a conflict when we consider variables with automatic storage duration found within a kernel. ISO/IEC TR 18037:2006 prohibits the use of address space qualifiers on variables with automatic storage duration, implicitly making them members of the generic address space (ISO/IEC, 2006, p. 38). Meanwhile, the HSA execution model requires that automatic variables within a kernel should be members of the private segment. This would result in the generic address space mapping to both the global and private segments for kernel agents.

We resolve this by introducing the notion of an implicit address space for rvalues and lvalues with automatic storage duration, where the specific address space used is a property associated with the hosting function. When compiling for the host CPU these segments are defaulted to the generic address space. When targeting HSAIL, these defaults are modified such that both automatic storage duration lvalues and rvalues are allocated in the address space which corresponds to the private segment. These implicit address space allocations are applied in the compiler frontend.

Variables with automatic storage duration map down to an `alloca` instruction in LLVM Intermediate Representation (IR). LLVM does not provide support for address spaces on `alloca` instructions, instead implicitly assuming the use of the generic address space. The `alloca` instruction must therefore be modified to add address space support.

We can illustrate this behaviour with an example. Listing 5-14 shows a simple function which stores values into two variables, The first, `a`, is a global variable and has static storage duration, while `b` has automatic storage duration.

If we use our compiler to compile this function for the host processor, then the generated LLVM IR appears as shown in listing 5-15. The `alloca` instruction returns an address in the generic address space and so both `a` and `b` can be viewed as being allocated within HSA's global segment.

```
// A global variable (static storage duration).
int a;

extern "C" void f() {
  a = 1;

  // A local variable (automatic storage duration).
  int b = 2;
}
```

**Listing 5-14:** Variable Storage Duration in C++

```
target triple = "x86_64-unknown-linux-gnu"

@a = global i32 0, align 4

; Function Attrs:
define void @f() {
entry:
  %b = alloca i32, align 4
  store i32 1, i32* @a, align 4
  store i32 2, i32* %b, align 4
  ret void
}
```

**Listing 5-15:** Variable Storage Duration in LLVM IR for Host Processor

The same code compiled to target a kernel agent generates the LLVM IR shown in listing 5-16. The function `f` now carries an additional attribute, `stk_as 1` indicating the address space that should be used for allocations with automatic storage duration. This annotation is necessary to ensure that any further transformation or optimization passes which generate further allocations are able to place them in the correct address space.

```
target triple = "hsail64-unknown-linux-gnu"

@a = global i32 0, align 4

; Function Attrs: stk_as 1
define void @f() stk_as 1 {
entry:
  %b = alloca addrspace(1) i32, align 4
  store i32 1, i32* @a, align 4
  store i32 2, i32 addrspace(1)* %b, align 4
  ret void
}
```

**Listing 5-16:** Variable Storage Duration in LLVM IR for HSA Kernel Agents

We can see that `alloca` and corresponding `store` instructions now operate on named address space 1. This corresponds to HSA's private segment. The global variable, `a`, remains in the generic address space and HSA's global segment.

The use of address space annotations at both the function and the instruction level may appear to be an unnecessary duplication of information. This is necessary due to the structure of the LLVM code generation framework. Prior to the addition of our extensions, LLVM already required instruction level address space annotations wherever operands reference a non-default address space. Our extensions simply ensure that this behaviour is replicated to correctly propagate this information from our extended `alloca` instructions. However, some LLVM optimization passes will introduce new allocations into an existing function. In this case, we must ensure that these new allocations also make use of the correct address space. The function level annotation acts as a reference point for this information.

### 5.4.3 Scoped Atomic Operations

HSA defines a set of atomic operations on addresses in the group and global segments, and on flat addresses corresponding to those two segments.

The set of atomic operations in HSA are a superset of those required to implement the C11/C++11 atomic functions. In addition to the memory order parameter that we see in C++11 atomics, atomic instructions in HSAIL also include memory segment and scope components.

The scope component can be used to limit the communication overhead of synchronization. For example, an application might choose to use an allocation in the global segment, but only require the side-effects of a particular atomic to be visible to work-items in the same work-group.

The HSAIL LLVM backend initially lacked support for both atomic operations and signals, making it impossible to implement data structures which support safe concurrent access from multiple agents. Support for these instructions was therefore added to the compiler backend.

In order to maintain compatibility with existing C++ code, while still exposing the full capabilities of HSA, we take two separate approaches to supporting atomic operations in our compiler and runtime.

The first is to implement lowering from LLVM's atomic instructions to equivalent HSAIL instructions. This ensures that all existing code which relies upon atomic operations can be compiled without modification and continues to function correctly. The C++11 atomic functions, and therefore the LLVM instructions, do not express the concept of memory scope on an atomic operation. Consequently, we must infer a scope to apply when generating the HSAIL instructions. For allocations in the global segment, and for flat addresses, `system` scope is selected. This is the most expensive scope, but ensures correct behaviour in all cases. For allocations in the group segment, we can be less conservative. By definition an allocation in the group segment is not accessible except by work-items within the same work-group, and so we can limit the scope to `wg` (work-group).

Whilst the approach described above is sufficient to successfully compile existing code that utilizes atomics, it suffers from some limitations. Firstly, there is no way to express scoped atomics with non-default scope. As described previously, system scope was selected as the default. This is the most conservative choice and

also the most expensive in terms of performance. Secondly, HSAIL introduces a small number of atomic operations which are not covered by the C++11 `std::atomic` class. These include `min`, `max`, `wrapinc`, and `wrapdec` operations. To resolve this, the compiler has been extended to expose a set of built-in functions that express all of the HSAIL atomic functionality. These are used to implement an alternative C++ template class for atomics, `rt::atomic`. This class closely mirrors the interface to the existing atomic class in the C++ standard library, `std::atomic`, extended with the additional operations found in HSAIL and additional parameters for memory scope.

### 5.4.4 Vector Loads and Stores

Whilst using Offload to develop RTKit (chapter 6) we encountered a number of code patterns that generate a series of strided sequential store instructions. These patterns resulted in surprisingly poor performance. One possible solution to this is to coalesce such stores into fewer, larger stores.

In addition to scalar load and store instructions, HSAIL defines vector memory instructions. These instructions enable the loading of two, three or four contiguous scalar values from memory into an equivalent number of independent registers, or stores from two, three or four independent scalar registers into a series of contiguous memory locations. For example, `st_v4_global_u32` instruction will combine the contents of four 32 bit registers and perform a single 128 bit store to a global segment address. The HSAIL LLVM backend used as a component of this work did not support the generation of these instructions. Consequently it was necessary to add support, both for instruction generation and load/store combining transforms to generate them.

In order to evaluate the performance of such strided memory accesses, and of coalescing through the use of vector instructions, a 512 MiB buffer was allocated and populated with instances of a structure composed of 4 32-bit unsigned integers. 1000 iterations were recorded for both load and store benchmarks.

Store performance was evaluated by using a single work-item per structure instance (i.e. 32 million work-items). An immediate value of zero was written to every element of each structure. The generated HSAIL was modified by hand to control the instructions used to perform the write.

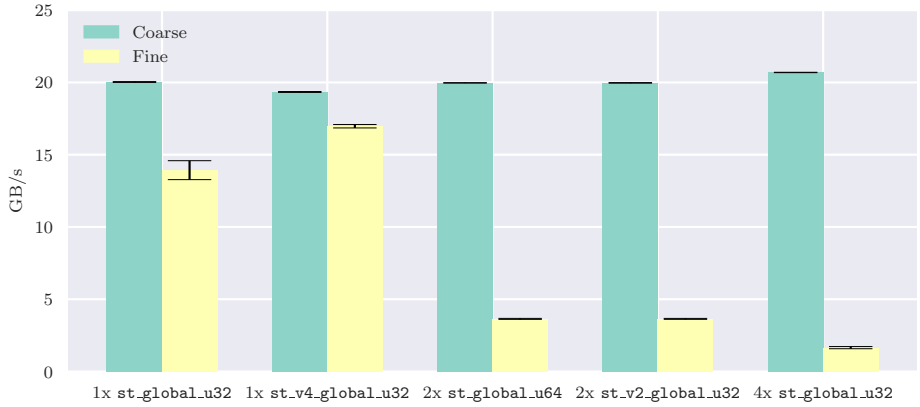



**Figure 5-2:** HSAIL Vector Store Performance

| **Store Instructions** | **Memory Granularity** | **Bandwidth (GB/s)** | |
|---|---|---|---|
| | | $\overline{x}$ | $s$ |
| 1 x `st_v4_global_u32` | Coarse | 19.34 | 0.03 |
| 2 x `st_global_u64` | Coarse | 19.97 | 0.01 |
| 2 x `st_v2_global_u32` | Coarse | 19.97 | 0.02 |
| 4 x `st_global_u32` | Coarse | 20.69 | 0.01 |
| 1 x `st_v4_global_u32` | Fine | 16.97 | 0.12 |
| 2 x `st_global_u64` | Fine | 3.64 | 0.03 |
| 2 x `st_v2_global_u32` | Fine | 3.64 | 0.03 |
| 4 x `st_global_u32` | Fine | 1.65 | 0.08 |
| 1 x `st_global_u32` | Coarse | 20.03 | 0.03 |
| 1 x `st_global_u32` | Fine | 13.93 | 0.66 |

**Table 5-1:** HSAIL Vector Store Performance

Figure 5-2 and table 5-1 illustrate the performance impact of vector stores. We found a small negative impact to coalescing store instructions when operating on coarse grained allocations. Conversely, we observe large gains to coalescing writes to fine grained allocations. Coalescing 4 contiguous writes using `st_global_u32` into a single `st_v4_global_u32` instruction results in a 10× increase in write bandwidth for this benchmark.

Unlike store performance, load performance cannot be trivially benchmarked in isolation. The HSA finalizer will optimize away any loads where the result of the load is not used in an expression with observable side-effects. To prevent these optimizations from removing loads, whilst minimizing the cost of additional work, we sum the 4 components of each structure, and write the result out to group memory.

In order to eliminate the potential overhead of a write to group memory, a second variant of this benchmark was also developed, where the write is enclosed in a conditional branch. If we make this branch conditional on the summed components of the input structure, and control the inputs to ensure that the branch is never executed, we can replace the cost of a write to group memory with a comparison and conditional jump instruction. This variant did not result in a measurable performance difference from the original benchmark, and so we can conclude that the a single 32-bit write to group memory has negligible cost when compared to 4 32-bit loads from global memory.

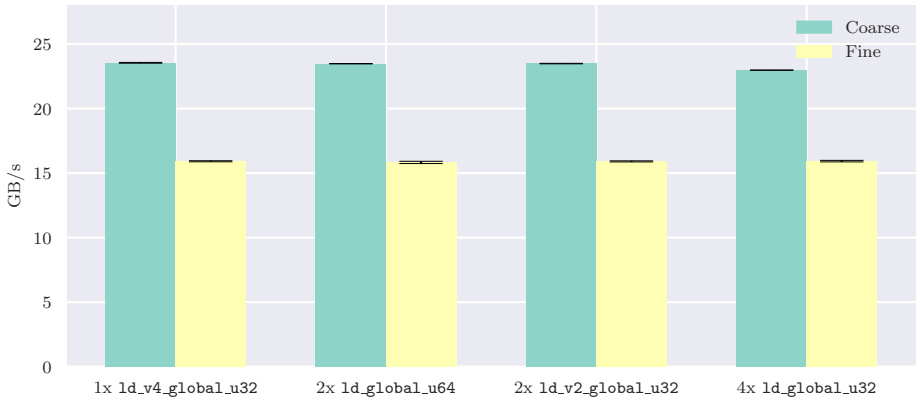



**Figure 5-3:** HSAIL Vector Load Performance

Vector load performance is summarized in figure 5-3 and table 5-2. Load coalescing provides a small performance benefit for coarse grained allocations, but no measurable benefit for fine grained allocations. The compute units in the Kaveri Accelerated

| Load Instructions | Memory Granularity | Bandwidth (GB/s) | |
|---|---|---|---|
| | | $\overline{x}$ | $s$ |
| 1 x `ld_v4_global_u32` | Coarse | 23.55 | 0.03 |
| 2 x `ld_global_u64` | Coarse | 23.48 | 0.01 |
| 2 x `ld_v2_global_u32` | Coarse | 23.49 | 0.02 |
| 4 x `ld_global_u32` | Coarse | 22.98 | 0.01 |
| 1 x `ld_v4_global_u32` | Fine | 15.92 | 0.04 |
| 2 x `ld_global_u64` | Fine | 15.83 | 0.09 |
| 2 x `ld_v2_global_u32` | Fine | 15.91 | 0.04 |
| 4 x `ld_global_u32` | Fine | 15.93 | 0.06 |

**Table 5-2:** HSAIL Vector Load Performance

Processing Unit (APU) each contain their own L1 cache, with 64 byte cache lines. Consequently, uncombined loads are still likely to be served from the cache.

## 5.5 RUNTIME LIBRARY

In addition to compiler support for our programming model, we also provide a runtime library. This library provides C++ classes and functions for the following:

- Built-in Kernel Functions
- Agents
- Queues
- Signals and Futures
- Scoped Atomics
- Memory
- Images and Samplers

In many cases, these classes provide dual implementations, such that they may be utilized in both host and kernel contexts.

### 5.5.1 Built-in Kernel Functions

HSAIL defines a number of instructions which have no analogous equivalent for host code. An example of this is the `workitemabsid` instruction, which returns the unique index of a work-item within the kernel dispatch grid. When targeting HSAIL, our compiler exposes a set of built-in functions corresponding to these instructions. Our runtime defines a set of C++ functions which correspond to these built-in functions when targeting HSAIL, and which result in undefined behaviour when called from outside a kernel.

### 5.5.2 Agents

As we discussed in section 2.7, HSA describes a heterogeneous system in terms of set of agents communicating through a shared memory system. Our runtime library provides functionality for enumerating agents and querying their capabilities and access to memory regions.

```
1  rt::agent *gpu_agent = nullptr;
2
3  // Enumerate all agents in the system, calling the lambda function
4  // for each one we find.
5  rt::enumerate_agents([&](rt::agent *agent) {
6    // Inspect the properties of the agent to verify that it is:
7    //   a) a GPU
8    //   b) a kernel agent
9    if (rt::device_type::gpu == agent->device_type() &&
10       agent->is_kernel_agent()) {
11     // Store the agent we found and cease enumerating agents.
12     gpu_agent = agent;
13     return false;
14   }
15
16   // Continue enumerating agents.
17   return true;
18 });
```

**Listing 5-17:** Agent Enumeration and Capability Introspection

Listing 5-17 provides a simple example of enumerating the agents within a system at runtime to identify a GPU agent capable of executing kernels.

### 5.5.3 Queues

Queues are the primary method for agents to communicate requests for work to be performed by other agents within the system. These queues are multi-producer queues, allowing for concurrent enqueuing of work from multiple agents or host threads. Queues may only be constructed from host code. However, packets can be added to the queue from both host and kernel code. This enables kernel agents to both enqueue additional work to themselves, providing support for dynamic parallelism, and to other agents. Additionally, we provide an interface to implement a software-backed queue capable of processing agent-dispatch packets.

```
// Create a software queue for processing agent dispatch packets on
// the host agent.
auto host_queue = host_agent->create_soft_queue(4096, callback);

// Create a hardware-managed queue for processing AQL packets on the
// GPU agent.
auto gpu_queue = gpu_agent->create_queue(4096);

// Enqueue a single work-item kernel to execute on the GPU and then
// wait for completion.
auto kernel_future = gpu_queue->parallel_for<class async_dispatch>(1,
  [&]() {
    // Send an agent-dispatch packet to the host CPU queue requesting
    // dynamic memory allocation and then wait for the AQL packet to
    // be serviced.
    void* ptr;
    auto host_future = host_queue->dispatch(MALLOC, &ptr, 1024);
    host_future.wait();

    // ptr now points to memory allocated by the host CPU. Kernel
    // execution can now continue.
  });

// Host waits for kernel completion.
future.wait();
```

**Listing 5-18:** Using a Host Queue to Provide Dynamic Memory Allocation to a Kernel Agent

Listing 5-18 and listing 5-19 provide an example of the use of queues to enqueue work between agents. HSA lacks the functionality to perform dynamic memory allocation directly from within a kernel. In this example, a kernel executing on a GPU makes a call back to the host processor to request dynamic memory allocation.

In Listing 5-18 we construct a software-managed queue running on the host agent, and a hardware-managed queue associated with a GPU agent. A kernel is enqueued to the GPU agent, which will in turn enqueue an agent-dispatch packet requesting dynamic memory allocation to be performed by the host agent.

```
// A callback function to handle agent-dispatch packets.
void callback(uint16_t msg, void **ret_address, uint64_t *args) {
  switch(msg) {
  case MALLOC:
    *ret_address = malloc(args[0]);
    break
  }
}
```

**Listing 5-19:** Agent Dispatch Packet Handler

Listing 5-19 illustrates a potential implementation of the callback function required to service agent-dispatch packets from a software-managed queue. Each agent-dispatch packet contains a 16-bit identifier intended to encode the requested function to perform, a return address which may optionally be set following packet execution and up to 4 64-bit arguments. These are mapped to the corresponding arguments of the callback function, which will be executed once for each agent-dispatch packet submitted to the software-managed queue.

### 5.5.4 Signals and Futures

HSA signals are light-weight primitives intended for notification and synchronization between different agents in an HSA system.

Signals are primarily intended for inter-agent synchronization, and not for fine-grained work-item synchronization within a single kernel dispatch. Consequently, operations on signals always behave like system-scope atomic operations. One notable difference between signals and atomic variables is that signals expose a wait operation. Whilst such a wait could be implemented as a spin-lock on an atomic variable, a signal wait operation may be implemented by hardware. This potentially allows an agent to enter a lower power state while waiting for a condition to be satisfied. This is their primary advantage over using system-scope atomics for synchronization.

We provide a signal class for system-wide communication and synchronization. This type is usable from both host code and kernels, and can be used as a primitive in the construction of more complex synchronization objects such as mutexes. Our compiler exposes a set of built-in functions for generating signal instructions when targeting HSAIL. Our runtime implementation maps signal operations to these built-in functions when targeting kernel agents, or to HSA runtime API functions when targeting the host processor. This is illustrated with a section of `signal::add` in listing 5-20.

```
1  void signal::add(int64_t value, memory_order order) {
2    switch (order) {
3    // ...
4    // Handle sequentially consistent acquire memory order case.
5    case scacq:
6  #ifdef __HSAIL__
7      // If we are compiling for HSAIL, call compiler builtin.
8      __hsail_signal_add_scacq(handle_, value);
9  #else
10     // If we are compiling for host, call HSA runtime API.
11     hsa_signal_add_scacquire(handle_, value);
12 #endif
13     break;
14   }
15 }
```

**Listing 5-20:** Host and Kernel Agent Implementations of `signal::add` Function

HSAIL lacks instructions to create or destroy signals. Instead, signals can only be created and destroyed through the use of HSA runtime API calls (`hsa_signal_create`, `hsa_signal_destroy`). Consequently signal variables may only be constructed in host code. All other operations, such as setting their value or waiting on a conditional state can be utilized from both the host processor and kernel agents.

All queue submission methods in our runtime, such as `parallel_for` or `barrier_and` return completion futures. These types build upon signals to enable querying execution status or waiting for completion of AQL packets. They also allow for the chaining of operations, potentially without host CPU intervention. Listing 5-21 illustrates this.

```
// Enqueue a kernel lambda to execute on a kernel agent and return
// immediately.
auto future = queue->parallel_for<class kernel>(1024, [&](){
  // Execute a kernel on the kernel agent...
});

// On kernel completion, execute a completion callback on the host
// processor.
future.then([&](){
  // Host lambda function.
});
```

**Listing 5-21:** Kernel Completion Callbacks

### 5.5.5 Memory

Full-profile HSA implementations must guarantee that memory allocated through standard system allocators such as `malloc` will be globally accessible with fine-grained coherency. Despite this, the HSA runtime provides its own specialized allocator. The use of this allocator may provide performance benefits for specialized use cases. For example, this allocator can be used to allocate memory with coarse-grained coherency, or to make use of the dedicated memory on a discrete GPU.

The pervasive availability of cache-coherent memory is both convenient and performant for many workloads, particularly due to eliminating the burden of explicit data movement. However, some memory-bound workloads may benefit from the reduced coherency or improved memory-agent locality offered by specific memory regions. In order to better support these workloads, we provide API functions for the introspection, allocation and deallocation of memory. Additionally, we provide functions for copying memory and transferring ownership of coarse-grained allocations between agents.

### 5.5.6 Images and Samplers

Much like other runtimes for heterogeneous computing, HSA provides a method to take advantage of GPU texture units. This is expressed through an optional image extension. Images are opaque objects tied to a single kernel agent and which do not participate in the HSA memory model.

We provide abstractions of both images and samplers to enable access to the performance benefits of texture hardware from within our programming model. Listing 5-22 provide an example of image usage, demonstrating the construction of a 4-channel read only image, importing of data from a host buffer and reading from within a kernel.

```
// Create a read-only 4 channel floating point 1K x 1K image on the
// agent.
const size_t WIDTH = 1024;
const size_t HEIGHT = 1024;
const size_t STRIDE = sizeof(float) * 4;
auto image = agent->create_image<rt::ro, rt::image2d, float>(WIDTH,
                                                             HEIGHT,
                                                             4);
// Populate the image from a buffer.
image.import(data, WIDTH * STRIDE,
                   WIDTH * HEIGHT * STRIDE,
                   0, 0, 0, WIDTH, HEIGHT, 1);

// The image is now populated. Run a kernel that loads from the image
// back into an array.
agent->parallel_for<class image_kernel>(1, [&]() {
  // Read from a 2D coordinate.
  rt::float4 vec = image.read(0.5f, 0.9f);

  // Access channels using vector swizzle syntax i.e. vec.xyzw
});
```

**Listing 5-22:** Using Images

### 5.5.7 Finalization

In section 5.3, we saw how our compiler identifies regions of code to be compiled for execution on kernel agents and compiles these regions of code to HSAIL. However, the kernel agents do not execute HSAIL directly. Instead, HSAIL must be further transformed into the native instruction sets of the kernel agents present in a system. The HSA specifications refer to this process as finalization (HSA Foundation, 2016b, p. 50). Since the presence of a particular kernel agent can only be determined at run time, this finalization must necessarily happen at run time.

At initialization, our language runtime implementation must complete a number of additional steps to ensure that native executable representations of the kernels are available to every kernel agent, and that all referenced global variables are correctly initialized.

We begin by querying the HSA runtime API to obtain a list of kernel agents in the system and then querying the ISA of each kernel agent. This gives us a list of ISAs for which we must generate native executable representations.

As described in the preceding sections, each translation unit in the application containing a kernel function was compiled to BRIG and embedded as an ELF section within the program executable. We examine the program binary and identify all of the sections containing BRIG code. Each of these BRIG objects is loaded into an HSA module and then combined to form an HSA program object. A program object is a collection of device agnostic modules. Once all modules are loaded, we utilize the HSA runtime to finalize the program object once per ISA, producing a set of code objects. Code objects are ISA-specific representations and contain the result of lowering from HSAIL to a native representation. In our case we have a single code object per ISA, containing all of the kernels and indirect functions for an application.

The final step in the finalization process is to patch the addresses of any global variables referenced from within the kernels to correspond to the addresses of the same variables within our host code. We enumerate all of the symbols contained in the code object, identifying any global variables referenced in kernels. For each global variable found, we retrieve the mangled name and search the host executable for a corresponding definition. The code object is then loaded into an HSA executable object. Each global variable in the HSA executable object is then assigned the corresponding address from the host representation and the HSA executable is frozen. This results in a final, immutable representation of the kernel executable.

This completes the compilation process. However, we still require a method of identifying individual kernels so that they can be dispatched as required. To do this we build a dispatch table per ISA. These tables are indexed by host function pointer. They contain the kernel handles and additional memory and alignment restrictions for invoking a kernel. We enumerate all of the symbols within the kernel executable, retrieve their mangled name and then search the host executable for a correspondingly named symbol. Where a match is found, an entry is added to the dispatch table.

At the conclusion of the finalization process, we have a set of natively executable representations of all of our kernels in the each ISA supported by the kernel agents in our system. Additionally, we have a corresponding set of tables which enable us to map from a host function pointer to an ISA specific kernel address. This information is sufficient to allow us to begin dispatching kernels to any kernel agent in our heterogeneous system.

## 5.6 COMPARISON TO EXISTING MODELS

In this section we will explore the differences between the model described in this chapter and other existing models for targeting heterogeneous systems. We provide detailed comparisons to SYCL, CUDA and Heterogeneous Compute (HC). We choose to focus on these three due to a combination of similar ancestry (HC, SYCL) and adoption rate (CUDA). Specifically, we compare to HC 0.9, SYCL 1.2 and CUDA 7.5. Further high-level comparisons to less widely used or more distantly related models can be found in section 3.2.

We will use a simple vector addition kernel as the basis of our comparison. However, we extend this to include a call to a device function in order to illustrate the required annotations in CUDA and HCC.

Listing 5-23 shows a simple vector addition kernel expressed in CUDA, with the addition itself moved into a separate function for illustrative purposes.

The example declares three vectors (`host_a`, `host_b` and `host_c`) on the host, and then executes a kernel to perform an element-wise sum of two of the vectors and stores the result in the third vector. Initialization of the contents of these vectors has been omitted for brevity.

From this example we can observe that CUDA is unable to operate on the C++ `std::vectors` directly. There are two reasons for this: CUDA's memory model requiring that data be copied into GPU device memory, and CUDA's inability to call the vector subscript operator.

The function `add` must be marked with an additional annotation, `__device__`. In cases where a function is intended for use on both the host CPU and on an accelerator, this annotation may be extended to `__host__ __device__`. While this does enable the creation of functions which are common to both contexts, porting existing code to

```cpp
#include <vector>

// A CUDA device function.
__device__ float add(float a, float b) {
  return a + b;
}

// CUDA kernel. Each thread calculates one element of c.
__global__ void vector_add(float *a, float *b, float *c) {
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  c[id] = add(a[id], b[id]);
}

int main(int argc, char* argv[]) {
  // Size of vectors
  const size_t SIZE = 1024;
  const size_t SIZE_IN_BYTES = SIZE * sizeof(float);
  // Host input and output vectors.
  std::vector<float> host_a(SIZE);
  std::vector<float> host_b(SIZE);
  std::vector<float> host_c(SIZE);
  // Allocate device vectors.
  float* device_a, *device_b, *device_c;
  cudaMalloc(&device_a, SIZE_IN_BYTES);
  cudaMalloc(&device_b, SIZE_IN_BYTES);
  cudaMalloc(&device_c, SIZE_IN_BYTES);
  // Copy host vectors to device.
  cudaMemcpy(device_a, host_a.data(), SIZE_IN_BYTES,
             cudaMemcpyHostToDevice);
  cudaMemcpy(device_b, host_b.data(), SIZE_IN_BYTES,
             cudaMemcpyHostToDevice);
  // Execute the kernel.
  vector_add<<<SIZE, 256>>>(device_a, device_b, device_c);
  // Copy array back to host.
  cudaMemcpy(host_c.data(), device_c, SIZE_IN_BYTES,
             cudaMemcpyDeviceToHost);
  // Free device vectors.
  cudaFree(device_a);
  cudaFree(device_b);
  cudaFree(device_c);
  return 0;
}
```

**Listing 5-23:** Implementing Vector Addition using CUDA

CUDA requires the introduction of these annotations throughout the source code. This is particularly undesirable in the case of third-party source code such as the C++ standard library.

Listing 5-24 shows the same vector addition kernel implemented in SYCL. Like CUDA, SYCL is also unable to operate on the host vectors directly. SYCL does not require the additional function annotations that we see in CUDA. This eases porting existing C++ code to SYCL. However, SYCL places restrictions on the types which are valid for use within a kernel. Due to the disjoint address spaces and lack of SVM in the underlying OpenCL implementations, SYCL 1.2 is unable to dereference the host pointer contained within the vector class.

The CUDA example in listing 5-23 required explicit allocation of GPU memory and copying of the vector contents. SYCL abstracts this through a system of buffers and accessors, which will provide efficient and automatic scheduling of memory transfers.

SYCL allows for the use of separate compilers for host and device code. To ensure layout consistency between compilers, SYCL restricts the set of valid types which may be passed between host and device code. SYCL requires that objects passed between host and device code are standard-layout types (ISO/IEC, 2014, p. 71, 215). This constraint prohibits many forms of C++ class and struct types, such as those with virtual functions, data members with differing levels of access control, or some forms of inheritance. Our model relies on the use of a common compiler for both host and device code, and ensures ABI compatibility between host and agent code. As such, we place no constraints on data types.

HCC represents the closest work to our programming model. HCC provides a C++ programming model which extends C++ AMP with additional capabilities to support HSA. HCC is a work in progress, and currently lacks a formal specification. Therefore, we compare our work to HCC 0.9. Due to the lack of documentation, the examples and all discussion of HCC's capabilities are derived from our experimental exploration. Similarly, we will base our discussion of HCC's features and constraints on the C++ AMP specification from which it is derived, and highlight the cases where HCC diverges from that specification.

C++ AMP requires function annotations on all device functions, similar to those found in CUDA. Like CUDA, C++ AMP provides a form of annotation suitable for use on both the host CPU and accelerator devices: `restrict(amp, cpu)`. HCC replaces the `restrict(amp)` annotation with a more modern C++11 generalized attrib-

```cpp
#include <vector>
#include <CL/sycl.hpp>

// A function which will be duplicated for the accelerator.
float add(float a, float b) {
  return a + b;
}

int main(int argc, char* argv[]) {
  // Size of vectors.
  const size_t SIZE = 1024;

  // Host input and output vectors.
  std::vector<float> host_a(SIZE);
  std::vector<float> host_b(SIZE);
  std::vector<float> host_c(SIZE);

  // Create a SYCL queue on the default device.
  cl::sycl::queue q;
  {
    // Allocate SYCL buffers, taking ownership of data.
    cl::sycl::buffer<float> buffer_a(host_a.data(), SIZE);
    cl::sycl::buffer<float> buffer_b(host_b.data(), SIZE);
    cl::sycl::buffer<float> buffer_c(host_c.data(), SIZE);
    q.submit([&](cl::sycl::handler& cgh) {
      using access_mode = cl::sycl::access::mode;
      auto device_a = buffer_a.get_access<access_mode::read>(cgh);
      auto device_b = buffer_b.get_access<access_mode::read>(cgh);
      auto device_c = buffer_c.get_access<access_mode::write>(cgh);

      // SYCL kernel. Each thread calculates one element of buffer c.
      cgh.parallel_for<class vector_add>(cl::sycl::range<1>(1024),
        [=](cl::sycl::id<1> id) {
          device_c[id] = add(device_a[id], device_b[id]);
        }
      );
    });

    // Buffers are destroyed. Results are written back to buffer c.
  }
  return 0;
}
```

**Listing 5-24:** Implementing Vector Addition using SYCL

```cpp
#include <vector>
#include <hc.hpp>

// A HCC device function. The [[hc]] annotation is mandatory in this
// case.
float add(float a, float b) [[hc]] {
  return a + b;
}

int main(int argc, char* argv[]) {
  // Size of vectors
  const size_t SIZE = 1024;

  // Host input and output vectors.
  std::vector<float> host_a(SIZE);
  std::vector<float> host_b(SIZE);
  std::vector<float> host_c(SIZE);

  // Execute the kernel.
  auto future = hc::parallel_for_each(hc::extent<1>(SIZE),
    [&](hc::index<1> id) [[hc]] {
      host_c[id[0]] = add(host_a[id[0]], host_b[id[0]]);
    });
  future.wait();

  return 0;
}
```

**Listing 5-25:** Implementing Vector Addition using HCC

ute, `[[hc]]`. This is a purely syntactic change. Furthermore, HCC attempts to relax the requirement that all device functions carry this annotation by inferring its presence. This relaxation appears to be work in progress. At present these annotations are still required in many cases. It is likely that this will be resolved in time. However, at the time of writing, the `[[hc]]` attribute appears to retain the viral nature of its **`restrict`**`(amp)` predecessor. In the case of the example shown in listing 5-25, we found that annotations were required for the kernel lambda function, and for the function `add`. However, HCC was able to correctly infer that the vector subscript operator should be treated as a device function without requiring modification of the standard library source code. However, more complex examples such as the map lookup example previously shown in listing 5-13 failed to compile without modific-

ation. By contrast, such annotations are not required in our model, except on kernel functions.

Like SYCL, C++ AMP restricts the types that may be referenced within kernels. C++ AMP introduces the notion of an *amp-compatible* type. This encompasses a restricted subset of the C++ fundamental types which are valid within an *amp-restricted* function (Microsoft Corporation, 2013, section 2.4.1.2). C++ AMP further defines rules for *amp-compatible* compound types (Microsoft Corporation, 2013, section 2.4.1.3). These rules include requirements that compound classes must not have virtual member functions or virtual base classes; that all members variables are *amp-compatible* and prohibit the use of pointers as members variables. In practice HCC appears to relax some of these constraints, and the provided example does successfully compile and execute. However, the details of these changes are not documented and it is unclear what limits apply.

C++ AMP provided container classes, `array` and `array_view`, as an abstraction to hide the complexity of disjoint address spaces and provide automatic scheduling of memory transfers. HCC retains these classes. However, HCC extends C++ AMP's model to support access through raw pointers in the same manner as our model.

There are further advantages to our approach not illustrated in the example presented. This may be because these features rely upon functionality that is unique to HSA or that is currently not implemented in existing models.

These include function overloading and template instantiation based on segments, support for global variables, concurrent access to shared data structures from multiple agents and dispatching kernels and host CPU function calls from agents.

In both CUDA and HCC, address spaces (or segments in HSA nomenclature) are associated with variable declarations, but not with pointers to these variables. In these programming models, taking the address of an address space-qualified variable results in an unqualified pointer. The correct address spaces for pointers are then computed in the compiler backend, rather than expressed in the language and compiler frontend. This is one of the key differences from our model and precludes the use of address spaces for function overloading or template instantiation.

In general, SYCL prohibits the use of global or static storage duration variables within kernels. CUDA does allow for the use of global variables, including variables which are shared between the host processor and GPU through the use of the `__managed__` keyword. However, it does not support concurrent access from both processors. Both HCC and our model support global variables without the use of additional keywords.

Concurrent access is fully supported, assuming the correct usage of atomic operations or memory fences to protect against data races.

By taking advantage of the fine-grained coherent memory found in HSA and system-wide signals and atomic operations, we can construct data structures and algorithms which rely upon concurrent access from multiple agents. The shared ring buffer described in the introduction to this chapter is such an example. Similar data structures are difficult or impossible to implement in CUDA or SYCL due to a combination of disjoint address spaces and a lack of memory coherency between accelerators and the host CPU.

CUDA and OpenCL 2.0 provide functionality to enable kernels to launch grids of child kernels on the same accelerator device. SYCL 2.2 specifies similar functionality, although without an available implementation. Our model builds on HSA to provide a more flexible solution. A kernel or host agent may send AQL packets to any other agent in the system. In this way, agents are able to enqueue kernels to themselves or other kernel agents, or to call CPU functions through the use of a host queue and agent-dispatch packets. At the time of writing, HCC does not currently appear to offer equivalent functionality, although it is clearly implementable with further work.

## 5.7 EVALUATION

In this section we will examine a number of benchmarks in order to understand the performance characteristics of the compiler and runtime described in this chapter.

### 5.7.1 Evaluation Objectives

Our objective in this section is to establish a base level understanding of the performance of both HSA and our programming model, and to provide a comparison to OpenCL on identical hardware. We accomplish this through the use of a number of small benchmarks, selected to demonstrate a variety of different features and workloads.

This serves as validation of the functional correctness of our programming model, and the underlying HSA implementation.

```cpp
#include <vector>
#include <rt/compute/compute.h>

// A function which will be automatically duplicated for the device.
float add(float a, float b) {
  return a + b;
}

int main(int argc, char* argv[]) {
  rt::initialize();
  // Size of vectors
  const size_t SIZE = 1024;

  // Host input and output vectors.
  std::vector<float> host_a(SIZE);
  std::vector<float> host_b(SIZE);
  std::vector<float> host_c(SIZE);

  auto future = rt::parallel_for<class vector_add>(rt::throughput,
    SIZE, [&]() {
      // Get our global thread ID
      int id = rt::builtin::workitemabsid(0);
      host_c[id] = add(host_a[id], host_a[id]);
    });
  future.wait();

  rt::terminate();
  return 0;
}
```

**Listing 5-26:** Implementing Vector Addition using our C++ programming model for HSA

We do not attempt to address larger application-scale examples in this evaluation. This role is instead filled by RTKit, our ray tracing described in chapter 6. RTKit was developed concurrently to, and depends upon, the compiler and programming model described in this chapter.

### 5.7.2 Evaluation Plan

Due to a combination of our early access to HSA, and the lack of available high-level languages targeting HSA, no pre-existing benchmarks were available. We therefore selected a number of pre-existing OpenCL benchmarks and ported them to utilize our programming model. These benchmarks were selected to evaluate a range of different workloads and application characteristics, and so both demonstrate the functional correctness of our compiler and programming model, and provide a base level understanding of the relative performance of OpenCL and HSA.

We make use of six benchmarks, described below.

1. Kernel Dispatch
   - Verify HSA the purported latency advantages of user-mode queueing.
2. GPU-Stream
   - Establish achievable memory bandwidth for HSA and OpenCL.
   - Four computationally trivial kernels, launched individually.
3. Bitonic Sort
   - Memory-bound workload, with large number of data-dependent kernel dispatches.
4. Black-Scholes
   - Compute-bound benchmark.
   - Illustrates the potential advantages of fine-grained memory.
5. Discrete Cosine Transform
   - Compute bound benchmark. More computationally complex than Black-Scholes.
6. SVM Binary Tree Search
   - Irregular workloads and access patterns.

Our basic testing strategy is identical to that described in section 4.4, unless otherwise noted in the description of specific benchmarks. Where kernel timings are presented, they are calculated from OpenCL/HSA profiling events. Timings exclude loop overhead by timing the loop body. Minimum timer resolutions and precision are identical to those previously presented in section 4.4.4.

### 5.7.3 Experimental Setup

The benchmark system used in this evaluation is based around an AMD Kaveri APU, with CPU and GPU cores integrated into the same die and sharing the same Dynamic Random-Access Memory (DRAM). The processor is an AMD A10-7850K APU, containing four CPU cores and eight GPU compute units. The CPU cores have a base frequency of 3.70 GHz, while the GPU cores execute at 720 MHz. This machine has 16 GB of DDR3-1600 DRAM. This gives a peak theoretical bandwidth of 25.6 GB/s.

Our runtime and HCC both utilize the HSA 1.0.3 runtime. For OpenCL and SYCL-based benchmarks we use the OpenCL driver from the AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) v3.0 (version 1800.8) as the underlying OpenCL implementation.

### 5.7.4 Kernel Dispatch

We begin our benchmarks with a simple investigation into the relative cost of dispatching a kernel. Enqueuing a kernel and synchronizing with the host processor on completion has some overhead. This acts a lower bound on the size of workloads which are likely to result in performance improvements when offloaded to an agent within a heterogeneous system.

One of the stated benefits of HSA is that user-mode queueing reduces the cost of dispatching kernels. By reducing this fixed cost, finer-grained parallel workloads become viable for offloading.

We can validate this claim by using microbenchmarks to examine the relative costs of kernel dispatches. We aim to minimize the impact of executing a kernel and isolate the cost of queue processing and synchronization by dispatching an empty kernel with no arguments. We measure both the execution time required to submitting the

kernel into a queue and the total execution time required for the kernel to complete and synchronize with the host processor. If HSA does indeed deliver reduced latency on kernel dispatches, then we should observe lower completion durations.

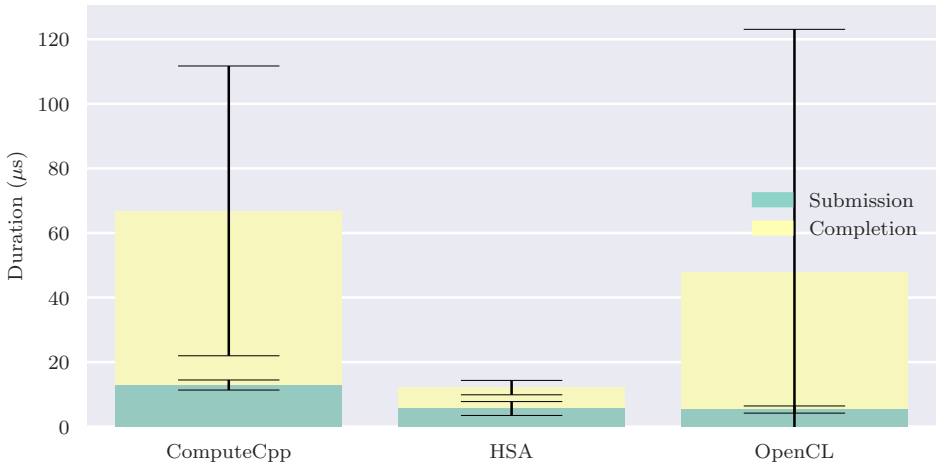



**Figure 5-4:** Kernel Dispatch Performance in SYCL, OpenCL and HSA

| **Runtime** | **Benchmark** | **Duration ($\mu$s)** | |
|---|---|---|---|
| | | $\overline{x}$ | $s$ |
| ComputeCpp | Submission | 12.90 | 1.56 |
| | Completion | 66.85 | 44.89 |
| HSA | Submission | 5.62 | 2.15 |
| | Completion | 12.10 | 2.23 |
| OpenCL | Submission | 5.30 | 1.11 |
| | Completion | 48.02 | 75.07 |

**Table 5-3:** Kernel Dispatch Performance in SYCL, OpenCL and HSA

From figure 5-4 and table 5-3, we observe greatly reduced kernel completion latency when using HSA and our runtime. This suggests that the use of user-mode queues in HSA does indeed result in lower kernel dispatch overheads, and a lower bound on the minimum work size that can be successfully accelerated on HSA. We also note that these latency reductions manifest as a reduction in the mean duration that a kernel dispatch spends in a work queue. The observable cost of host code adding work to a dispatch queue is approximately equivalent for HSA and OpenCL..

We also observe that whilst the dominant cost for kernel dispatch in SYCL can be attributed to the underlying OpenCL implementation, there are further costs imposed by ComputeCpp's scheduler and dependency tracker.

### 5.7.5 GPU-Stream

HSA's memory model is based entirely on shared virtual memory. This differs from OpenCL, where device memory allocations may use a separate address space. Fine-grained SVM also carries potential overheads due to the need to maintaining memory consistency across processors. In this benchmark, we aim to measure the achievable memory bandwidth of memory allocations in OpenCL and our HSA-based programming model, and to establish whether the use of SVM imposes significant performance costs.

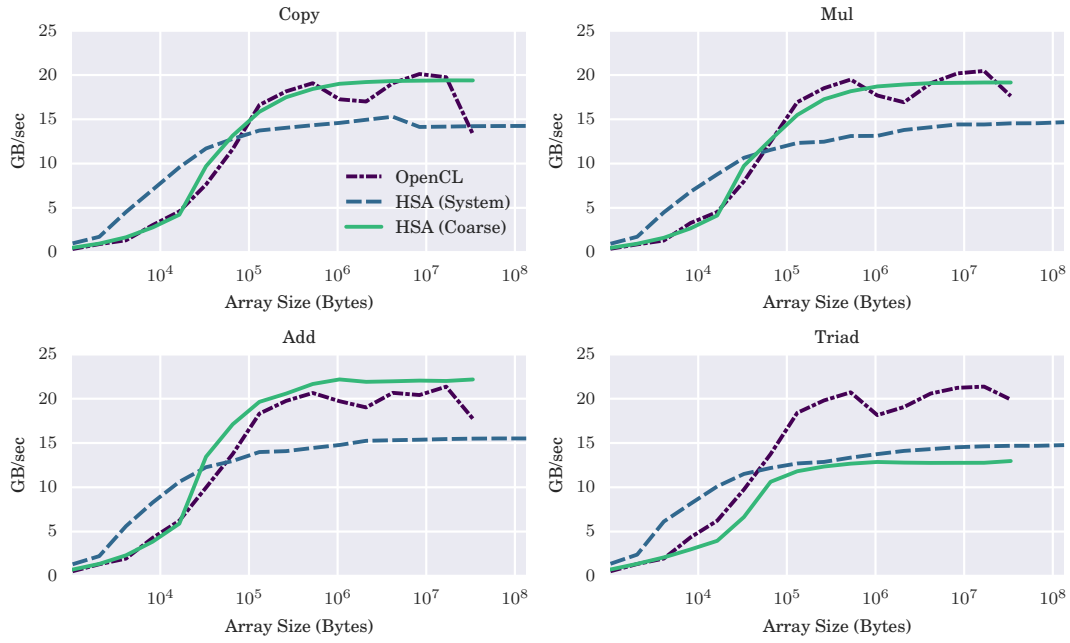



**Figure 5-5:** GPU-STREAM Memory Bandwidth Performance for OpenCL and HSA

In order to illustrate the performance characteristics of the memory on our test system, we use a modified implementation of the STREAM benchmark (McCalpin, 1995). The STREAM benchmarks measure sustained memory bandwidth over four simple kernels. We extend GPU-STREAM[2] to make use of our compiler and runtime. GPU-

2 https://github.com/UoB-HPC/GPU-STREAM

STREAM measures the time required to enqueue a kernel and synchronize back to the host processor. The cost of copying the input and output arrays between the host processor and the GPU are excluded from the measurements.

The four kernels in the STREAM benchmark correspond to simple, element-wise operations on vectors. Given three vectors: **A**, **B** and **C**, and a scalar value: $s$, the four benchmarks implement the following operations:

- Copy: $\mathbf{C} = \mathbf{A}$
- Add: $\mathbf{C} = \mathbf{A} + \mathbf{B}$
- Scale: $\mathbf{C} = \mathbf{A} \cdot s$
- Triad: $\mathbf{C} = \mathbf{A} + \mathbf{B} \cdot s$

If the use of SVM imposes no additional performance costs, we should achieve equivalent bandwidth for both OpenCL and HSA, regardless of whether memory is allocated from a fine or coarse-grained memory region.

From figure 5-5, we can observe that coarse-grained memory in HSA and OpenCL buffers produce similar performance for the Copy, Mul and Add kernels, although coarse-grained memory produces a more predictable performance curve. We also observe that using fine-grained system memory appears to have a lower overhead when dispatching kernels, and so results in improved performance for buffers below approximately 64 kB. However, fine-grained memory results in significantly reduced bandwidth on larger data sets. The OpenCL and HSA runtimes have a maximum size of 256 MB for OpenCL buffers and coarse-grained allocations respectively, while fine-grained allocations are only constrained by available DRAM.

In examining the OpenCL results, we observe performance drops at some buffer sizes. This behaviour is also reflected in several upcoming benchmarks. The allocation of these buffers is the responsibility of the OpenCL driver, and the internal memory allocation strategies used represents a black box to us. One reasonable hypothesis is that these performance drops reflect a change in allocation strategy, such as a switch from pinned to unpinned memory.

We also note that the Triad kernel generates unexpectedly poor performance for coarse-grained allocations. The coarse and fine-grained measurements were performed using the same finalized kernel. The only difference in this case is the manner in which the memory was allocated. In this case, coarse-grained memory fails to

achieve the throughput of fine-grained system memory, and performs significantly worse than OpenCL buffers.

The coherence granularity of a memory allocation used as a kernel argument is only available at kernel dispatch time, and so does not affect either the HSAIL generated by our compiler, or the finalization step within our runtime. As such, it is unclear why the performance of coarse-grained allocations in the Triad example differs from the pattern observed in the preceding examples, and can only be attributed to interactions within the driver and hardware itself.

### 5.7.6 Bitonic Sort

Batcher's bitonic mergesort (Batcher, 1968) is a parallel sorting algorithm. A bitonic mergesort uses repeated sequences of pairwise comparisons to merge bitonic sequences of increasing size, until the size of the sequence grows to encompass the complete set of elements. Batcher's mergesort is relatively simple to implement on a GPU, but requires a large number of kernel dispatches, and consequent intermediate memory stores, due to execution model constraints.

We evaluated an implementation of Batcher's bitonic mergesort from the AMD APP SDK, and provide a comparison across a range of array sizes. Additionally, we provide results for `std::sort` from the C++ standard library. Bitonic sorting requires multiple kernel executions to completely sort a dataset, and so we reduce our sample size to 100 iterations of the full sorting algorithm. Throughput is computed by timing the total wall-clock time elapsed between dispatching the first kernel launch, and a synchronizing wait for the completion of the final kernel execution. Data movement to and from the GPU is excluded from the timings. Batcher's mergesort has constant complexity regardless of whether the input data is pre-sorted. Despite this, we use an input buffer of randomly generated numbers. The results can be found in figure 5-6. None of the results include the initial cost of populating the input buffer. We also note that the bitonic sort sample found in the AMD APP SDK, and consequently our port, is implemented for clarity and not optimized for maximum performance.

For small array sizes, both HSA implementations outperform the OpenCL implementation. However, these small workloads are the least appropriate to processing on the GPU, and the throughput of the CPU implementation outclasses all three GPU variants. For larger datasets($> 10^5$ elements), we observe that a GPU-based sort using our HSA-based model and coarse-grained memory becomes competitive with

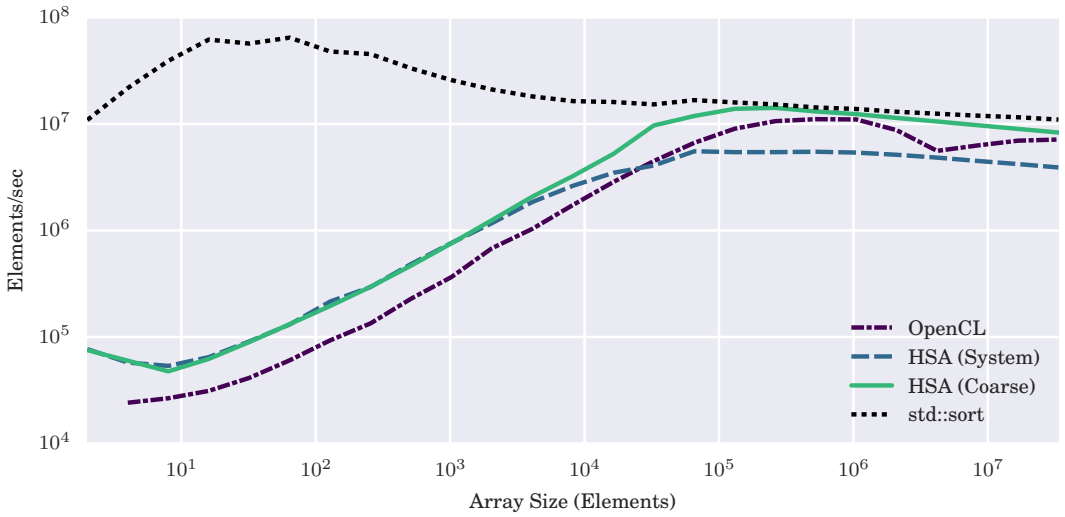



**Figure 5-6:** Sort Throughput Scaling - Bitonic Sort for OpenCL and HSA versus Sequential CPU Sort

`std::sort`. The coarse-grained HSA-based sort outperforms the OpenCL sort both in terms of throughput and consistency. However, we note that in the majority of cases, it appears that simply using std::sort may be both the simplest and more performant option here.

### 5.7.7 Black-Scholes

The GPU-STREAM and bitonic sort benchmarks are memory-bound use-cases. By contrast, the Black-Scholes option pricing model is commonly used as an exemplar compute-bound parallel workload. We provide a comparison of the Black-Scholes sample from the AMD APP SDK to a version ported to run on our HSA-based compiler and runtime. Given the use of identical hardware, and the compute-bound nature of the Black-Scholes algorithm, we should anticipate equivalent kernel performance between OpenCL and HSA implementations, regardless of the use of fine or coarse-grained allocations. Any discrepancies in kernel execution times must therefore be attributed to either compiler maturity, work-item scheduling strategies within the runtime, or additional overheads imposed by the requirements of the specific runtime. In additional to measuring kernel execution times, we also measure total execution time, inclusive of data movement to and from GPU accessible buffers. Given that Black-Scholes is a compute-bound workload, and our hypothesis of

equivalent kernel execution times, we should anticipate improved performance from fine-grained allocations here due to coupling the lack of data movement provided by fine-grained SVM with a kernel that is insensitive to memory bandwidth.

The Black-Scholes kernel relies upon the OpenCL built-in functions for exponential and logarithmic functions. As described in section 2.7, HSA lacks a maths library. Whilst HSAIL does provide equivalent instructions to many of the OpenCL built-in functions, it does not provide base $e$ exponentials or logarithms. In order to resolve this, we provided our own implementations of `exp` and `log`, adhering to the same precision requirements as described in the OpenCL specification ($\leqslant$ 4 Unit of Least Precision (ULP)). Beyond providing the necessary implementations of `exp` and `log`, the port makes no algorithmic changes to the kernel being profiled.

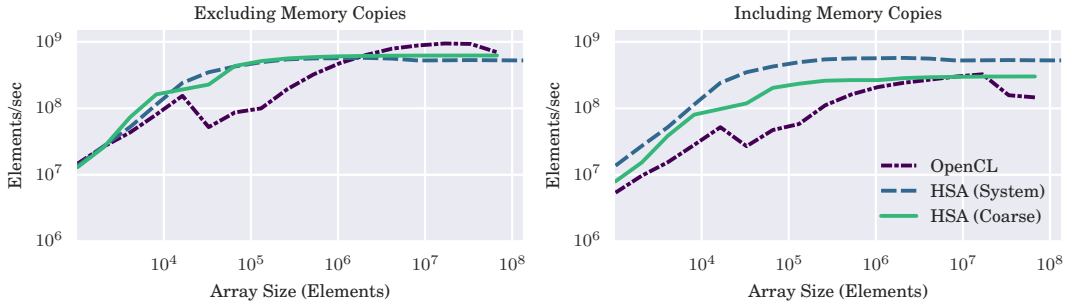



**Figure 5-7:** Black-Scholes Throughput Scaling for OpenCL and HSA

We measure execution time, both inclusive and exclusive of memory copies. Our results can be seen in figure 5-7. Ignoring the cost of memory transfers, the OpenCL kernel achieves higher peak performance on large datasets. We attribute this to a more mature compiler in the AMD OpenCL implementation, when compared to the combination of our compiler and the finalizer in the HSA runtime. Our HSA-based runtime benefits from faster kernel dispatches resulting in improved performance for small datasets, and also demonstrates a more stable and predictable performance curve throughout.

When the cost of memory transfers are considered, the fine-grained results are unchanged due not requiring copies. However, the throughput for both OpenCL and coarse-grained allocations is reduced. The net result is that whilst fine-grained allocations produce the highest kernel execution times, the elimination of copying still leads to the greatest total throughput.

### 5.7.8 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) encodes a sequence of points as a sum of cosine functions. It is commonly used in image and video compression. We ported the DCT sample from the AMD APP SDK. This sample performs an 8 by 8 DCT. We evaluated 1000 iterations of the DCT over 4 K resolution images ($3840 \times 2160$ pixels). Like Black-Scholes, the DCT benchmark is a compute-bound workload. As such we should anticipate equivalent performance for fine and coarse-grained HSA implementations, with the maturity of the finalizing compilers within the runtimes being the most likely root cause for any discrepancies in kernel execution times between OpenCL and HSA. The results of this evaluation can be found in table 5-4 and figure 5-8.

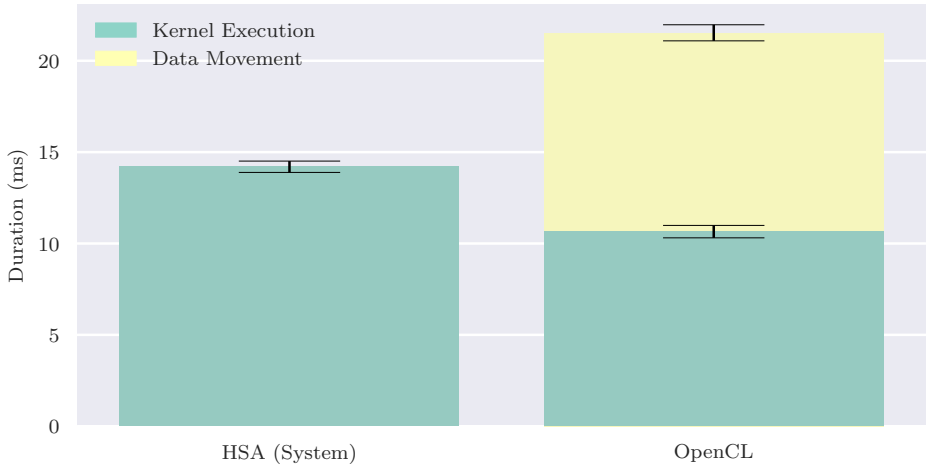



**Figure 5-8:** $8 \times 8$ Discrete Cosine Transform Performance for OpenCL and HSA at 4 K Resolution

| **Runtime** | **Kernel Execution** (ms) | | **Memory Transfers** (ms) | |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | $\overline{x}$ | $s$ |
| HSA (System) | 14.20 | 0.31 | | |
| OpenCL | 10.65 | 0.34 | 10.88 | 0.44 |

**Table 5-4:** $8 \times 8$ Discrete Cosine Transform Performance for OpenCL and HSA at 4 K Resolution

In terms of kernel execution, our runtime achieves 75% of the performance of that attained by the OpenCL runtime. However, we note that the use of HSA also enables us to avoid the memory copies required for OpenCL. The DCT kernel is arithmetic-

bound in both OpenCL and HSA, and is not limited by memory bandwidth on our evaluation hardware. As such, we observed no measurable performance difference between coarse and fine-gained memory allocations.

### 5.7.9 Shared Virtual Memory Binary Tree Search

For our final benchmark, we make use of a binary tree search. Unlike previous examples, which showcased highly regular workloads, this benchmark generates extremely irregular workloads and memory access patterns. The SVMBinaryTreeSearch sample from the AMD APP SDK serves to highlight one of the key advantages of our approach. This sample uses the support for coarse-grained shared virtual memory introduced in OpenCL 2.0 to implement a binary tree that is manipulated on both the CPU and GPU. Due to differing languages used for host code (C++) and device code (OpenCL C) in the sample, the data structures for representing the tree nodes and representing search keys are defined twice, in two different source files, but must maintain compatible binary layouts. Under our model, a single-source language and a single definition of data types can be used.

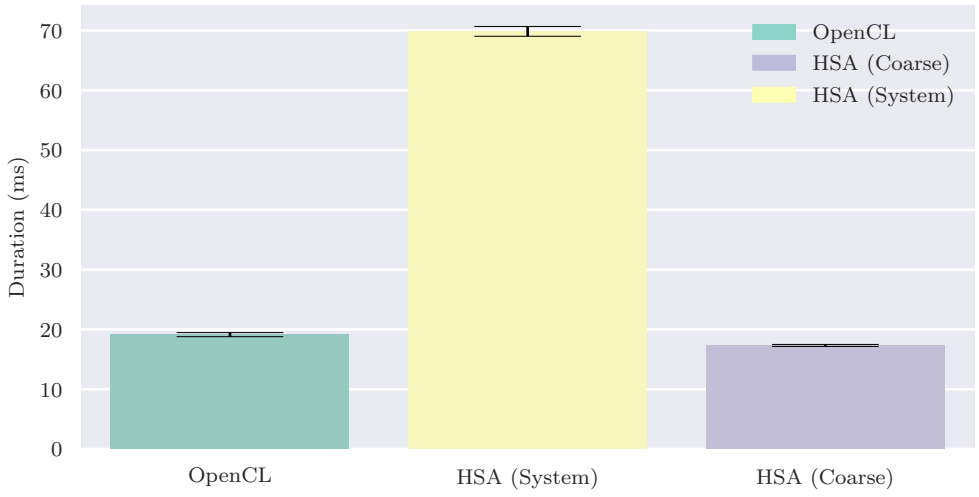



**Figure 5-9:** Shared Virtual Memory Binary Tree Search Performance for OpenCL versus Fine and Coarse-grained memory in HSA

For the evaluation, a set of 64 K random numbers are generated and a binary search tree is constructed from them. We then generate 1 million random search keys, and

| Runtime | Kernel Execution (ms) | | Fetch Size (kB) | Cache Hit Rate |
|---|---|---|---|---|
| | $\overline{x}$ | $s$ | | |
| OpenCL | 19.13 | 0.34 | 276,000 | 79.80% |
| HSA (System) | 69.84 | 0.82 | 12,000 | 33.60% |
| HSA (Coarse) | 17.32 | 0.17 | 278,000 | 79.50% |

**Table 5-5:** Shared Virtual Memory Binary Tree Search Performance for OpenCL versus Fine and Coarse-grained memory in HSA

perform a parallel search, matching nodes to corresponding search keys. The results of this evaluation can be found in figure 5-9 and table 5-5.

Similar to our evaluation of achievable memory bandwidth in the GPU-Stream benchmark, if there is no additional cost to utilizing fine-grained memory in HSA, we should observe equivalent performance for both OpenCL and both HSA variants.

In practice, we observe that using fine-grained (system-allocated) memory performs poorly for this use case. The HSA (Coarse) and HSA (System) results share the same kernel, but result in significant performance differences. Profiling the kernels reveals low cache hit rates (34%) coupled with a smaller quantity of data fetched from memory for the fine-grained allocation. The coarse-grained allocation achieves much a higher hit rate (80%), along with a greater quantity of data fetched. These two examples execute identical kernels, and so we conclude that these performance discrepancies are attributable to differing cache and memory management strategies for coarse and fine-grained allocations. The OpenCL implementation results in similar behaviour to the coarse-grained example. When comparing the performance of our runtime utilizing coarse-grained memory on HSA to the equivalent kernels running on OpenCL, we achieve a modest speedup whilst also demonstrating reduced performance variance.

## 5.8 LIMITATIONS AND FUTURE WORK

Whilst my C++ programming model already offers functionality that cannot currently be achieved today by models such as SYCL, there remains significant scope for improvement.

There are a number of constraints which currently prevent us from providing full support for C++. Most notably function pointers, and by implication virtual methods, are currently unsupported.

We encounter several challenges with respect to function pointers. Unlike OpenCL, HSA does provide a method for implementing indirect function calls, via the HSAIL `icall` instruction and so these features could plausibly be supported in the future. However, due to the order in which finalization occurs, functions called through a function pointer have a number of constraints which do not apply to functions called directly (HSA Foundation, 2016b, p. 267).

A further complication with function pointers, and especially virtual methods, derives from the process of using runtime finalization to generate native representations of functions for each agent in a system. By definition, this process results in multiple copies of functions, expressed in different ISAs. Since enumerating the agents in a system is performed at runtime, these copies are also generated at runtime. This in turn means that virtual function tables must both be populated at runtime and augmented with functionality to translate function pointers between agents.

The use of runtime type information such as the **`typeid`** or **`dynamic_cast`** operators on polymorphic types are similarly dependent on virtual function tables and are not supported.

The model described in this thesis also prohibits the use of C++ exceptions in kernels. Exceptions are challenging to define for parallel accelerators. This is particularly true for Single Instruction, Multiple Thread (SIMT) architectures, where multiple work-items share a program counter. Menon, De Kruijf and Sankaralingam (2012) describes some of the challenges presented by exceptions on GPUs, and propose a solution which requires hardware and compiler co-design.

Whilst we have been able to demonstrate the benefits of a single-source compiler and runtime for HSA, there remains significant scope for further benchmarking. One potential strength of HSA is to exploit more fine-grained parallelism. Rodina (Che, Boyer et al., 2009), SHOC (Danalis et al., 2010) and samples from the NVIDIA CUDA and AMD APP SDKs are commonly used as benchmarks in heterogeneous computing. However, these benchmarks primarily focus on discrete GPUs, and suffer from limitations when applied to our model. There currently appears to be a lack of suitable benchmark suites for the evaluation of shared memory heterogeneous processors such as System-on-Chips (SoCs) and APUs.

As a by-product of the compilation model, the Offload compiler generates a host-side representation of kernel functions which is currently unused. This is representation is typically incomplete. The many of the built-in functions such as `rt::workitemabsid` are simply stubs. However, by providing a CPU implementation of these functions, and an underlying scheduler, the same source code could be utilized to generate CPU code. If such an approach were combined with a vectorizing compiler, it may provide an efficient fallback in cases where kernel agents are not available, increasing the portability of our model.

Further work also continues on the compiler toolchain. Simon Brand has provided DWARF output for the generated HSAIL, enabling heterogeneous debugging support from LLDB (Brand, 2017).

## 5.9 DISCUSSION

In this chapter have we explored the design and implementation of a C++14 compiler and programming model for HSA. This model builds upon HSA's shared virtual address space to produce a single-source environment with tighter integration between host and device code than other models.

In introducing this thesis, we referred to three overarching themes:

- Providing early usage experience and validation to two new standards f or heterogeneous computing: SYCL and HSA
- The development and application of new C++ programming models for hete rogeneous computing.
- The use of those programming models to build domain-specific toolkits for problems in the field of visual computing.

This chapter has addressed the first two of those themes, with RTKit (chapter 6) building upon the compiler and programming model described in this chapter to satisfy the third.

In section 5.1, we discussed three design goals for our work on a C++14-based programming model for HSA:

- Enable validation of the specifications

- Directly expose new functionality
- Simplify experimentation

We have successfully implemented a single-source programming model utilizing HSA and demonstrated its use over a range of benchmarks. This demonstrates the suitability of the HSA platform for such endeavours. This in turn made it possible to contribute our experiences back to the HSA Foundation and better inform the design of future iterations of the specification.

Due to a need to be able to fully explore the properties of HSA, our model was designed to provide direct access to all of the features of HSA, rather than utilizing them as part of an implementation of an existing high-level runtime specification such as OpenMP or SYCL. We believe that our programming model and runtime library provides sufficient interfaces to allow access to all of the features of the HSA runtime library, and that the combination of our compiler and library of built-in functions exposes the full HSAIL instruction set.

The combination of a single-source programming model and the use of call-graph duplication to limit the use of non-standard keywords allows for tight integration between host and device code, and eases migration. This in turn simplifies experimentation and the porting of existing C++ code to execute on kernel agents.

Prior to beginning work on our programming model, there were no publicly available performance figures for any HSA implementation. We have demonstrated comparable performance to OpenCL across multiple benchmarks. In addition to validating the functional correctness of our compiler and programming model itself, our implementation and evaluation also serves to provide evidence of the suitability of HSA itself for building new parallel programming models.

Our work has revealed significant performance differences between coarse and fine-grained memory accesses. In general, coarse-grained regions offer higher bandwidth and is more forgiving of random access patterns. This is coupled with a need to explicitly transition ownership of data between agents. In general, coarse-grained memory offers similar costs and behaviour to OpenCL's buffers, albeit with more consistent scaling.

Fine-grained memory typically results in lower bandwidth and consequently increased kernel execution times in comparison to coarse-grained memory. However, in many cases this is mitigated by the removal of data transfers between agents. Fine-grained cache-coherent memory also allows for the implementation of data structures

which cannot otherwise be implemented. The shared ring buffer supporting concurrent access from the host processor and a kernel agent illustrated in listing 5-1 show is one such example.

These findings were generated using an AMD Kaveri-series processor. Representatives from AMD have suggested that the performance discrepancy is reduced in more recent Carrizo-series processors [personal communication, B. Sander, 2015]. However, we were unable to procure suitably configured hardware to verify this claim during the course of the project.

In detailing the goals of our programming model, we stressed the need for a programming model which lowered the cost of experimentation when porting code between the host CPU and kernel agents. In the subsequent chapter, we will make practical use of our compiler and programming model in just such a manner. We will investigate strategies for accelerating ray tracing on a heterogeneous system, and will make use of our compiler and runtime as the basis of that work.

# 6 RTKIT – RAY TRACING ON HETEROGENEOUS SYSTEM ARCHITECTURE

Throughout the course of this thesis, we investigate approaches to exposing the performance of heterogeneous systems to domain experts in fields such as image processing or ray tracing. We described one possible approach to this in chapter 4, where we described a technique for building a domain-specific language for image processing upon SYCL. This approach was able to provide performance benefits over OpenCV, and deliver similar performance to Halide, on our evaluation system. However, due to SYCL's compilation model and to the static nature of expression templates, mapping to alternative architectures would require the intervention of a machine expert.

In the preceding chapter, we described our compiler and programming model for HSA. Whilst chapter 5 presented a number of benchmarks, these were primarily microbenchmarks based on single or small numbers of kernels. In this chapter, we will explore the design of RTKit, our new library for accelerating ray tracing on heterogeneous systems, which builds on the compiler and runtime described in chapter 5.

Both our work on Offload (chapter 5) and our work on RTKit began prior to any HSA runtime implementation or hardware becoming available. As such, the final performance characteristics of the evaluation platform were not well understood early in the development of RTKit. This makes the use of a relatively static approach, such as that provided by our DSL from chapter 4, inappropriate. Instead, we focus on providing a toolkit that allows for the evaluation of a variety of different mappings of tasks to PU, vectorization strategies, and acceleration structure traversal algorithms.

Ray and path tracing (Kajiya, 1986; Whitted, 1979) make up an important group of computer graphics algorithms, primarily used for the simulation of light transport. In 1986, Immel, Cohen and Greenberg (1986), and Kajiya (1986), independently described the rendering equation:

$$L_o(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + \int_\Omega f_r(\mathbf{x}, \omega', \omega) L_i(\mathbf{x}, \omega')(-\omega' \cdot \mathbf{n}) d\omega'$$

This complex integral states that the outgoing radiance from a single point $\mathbf{x}$, and in a given direction $\omega$, is related to the sum of the incident radiance from all points visible from $\mathbf{x}$. Solving this integral globally for all points in a virtual three-dimensional scene would enable us to calculate illumination at all points and so generate a realistic image. However, solving this integral analytically is not tractable in the general case, and so we must instead compute approximate solutions using techniques such as Monte-Carlo integration.

Ray and path tracing provide one method for calculating such an approximation, and in particular for resolving the question of which surface points are visible from a particular point, $\mathbf{x}$.

This chapter is not focused directly on the problem of solving the rendering equation, but rather on the acceleration of component parts of the algorithms used to approximate a solution. Tracing rays through a virtual scene in order to identify sets of visible points is a computationally intensive task, and considerable effort has been devoted to accelerating this problem. Works such as Embree (Wald, Woop et al., 2014) and OptiX (Parker et al., 2010; Robison, 2011) have provided optimized ray tracing kernels for specific architectures. RTKit instead aims to provide a flexible toolkit suitable for the optimization of ray tracing algorithms on a variety of heterogeneous PU.

We begin the chapter, in section 6.1, with a discussion of the motivation for the development of RTKit, and by defining some related research questions. This is followed in section 6.2 by a description of the design and implementation of RTKit. In section 6.3, we make use of RTKit to evaluate a number of ray tracing pipelines, mappings to PU, vectorization strategies and acceleration tree traversal algorithms. We discuss limitations and possible future extensions to RTKit in section 6.4. Finally, we conclude our discussion of RTKit in section 6.5.

## 6.1 MOTIVATION

In the context of this thesis, the work described in this chapter serves several purposes.

The primary motivation for this chapter is to explore the suitability of HSA and APUs to accelerating ray tracing. A significant body of work has been devoted to accelerating ray tracing on GPUs, and to efficient Single Instruction, Multiple Data (SIMD) acceleration on CPUs. With the exception of a recently released work by Barringer, Andersson and Akenine-Möller (2016), the applicability of APUs to these workloads has not been deeply explored. Looking forwards, a further motivation is to provide a platform that could potentially enable the exploration of programmable ray tracing on future embedded and mobile SoCs that provide HSA support.

An additional contribution of this work is that it enables us to perform a comparison of CPU vs. GPU-based ray tracing *utilizing a common memory system*. Many researchers have demonstrated excellent ray throughput on discrete GPUs, where wide buses and high bandwidth memory are available. However, in this work we are able to explore performance where the peak memory bandwidth for both devices is identical due to a shared memory subsystem.

In motivating this thesis, we asserted both that heterogeneous systems have lead to a rise in complexity for software developers, and that high-level languages such as C++ could aid in tackling this complexity. This chapter serves as a validation of those claims. Throughout the course of this chapter, we will illustrate how the performance of workloads can vary based on the executing PU; the layout of data structures; the coherency properties of memory; and properties of the input data.

Given this complexity, optimizing for heterogeneous systems is a challenging task, with many competing factors. RTKit is based on a runtime graph, where nodes represent computational tasks. By manipulating the mapping of these nodes to particular PU, we can rapidly explore the performance implications of each mapping. The use of a high-level language supporting template metaprogramming, and a single-source programming model, greatly eases this process by enabling us to develop generic implementations of tasks, while retaining the ability to exploit PU-specific optimizations such as SIMD vectorization.

Furthermore, this chapter provides a larger case study on the practical application of our HSA-based C++ programming model, previously described in chapter 5. This serves to provide further validation of our approach.

With these points in mind, we can define three research questions:

- Is our programming model sufficient to implement a ray tracing pipeline?
- Can such ray tracing pipelines be implemented in a heterogeneous manner?
- Does doing so provide a performance benefit?

## 6.2 DESIGN AND IMPLEMENTATION

We can conceptually decompose ray tracing algorithms into a series of stages: ray generation, ray-geometry intersection testing, shading, and the accumulation of radiance into a final output image.

In the simplest of algorithms, such as Appel's ray casting algorithm (Appel, 1968), these stages can be arranged to form a simple linear pipeline. More complex examples such as Whitted ray tracing (Whitted, 1979) or path tracing (Kajiya, 1986) are recursive, and so form a cyclic graph.

Our framework expresses ray tracing algorithms as graphs where nodes represent computational tasks, and edges represent data dependencies. These nodes may be implementations of the previously mentioned stages in a ray tracing algorithm. Alternatively, they may be additional tasks introduced for performance optimization reasons, such as the compaction of sparse data; or house-keeping tasks required by the framework such as tracking statistics or scheduling data movement between regions of coarse-grained memory.

By controlling which HSA agent a particular node is executed on, we can explore the potential benefits of heterogeneous systems to ray tracing.

RTKit is a large software project, and is organised into five layered software libraries, as illustrated in figure 6-1.

### 6.2.1 libCompute - Heterogeneous Compute Runtime

The foundations of RTKit are provided by the compute support library, libCompute. This is the runtime library supporting our C++ programming model. Much of this library has previously been described in chapter 5. It offers abstractions over the HSA

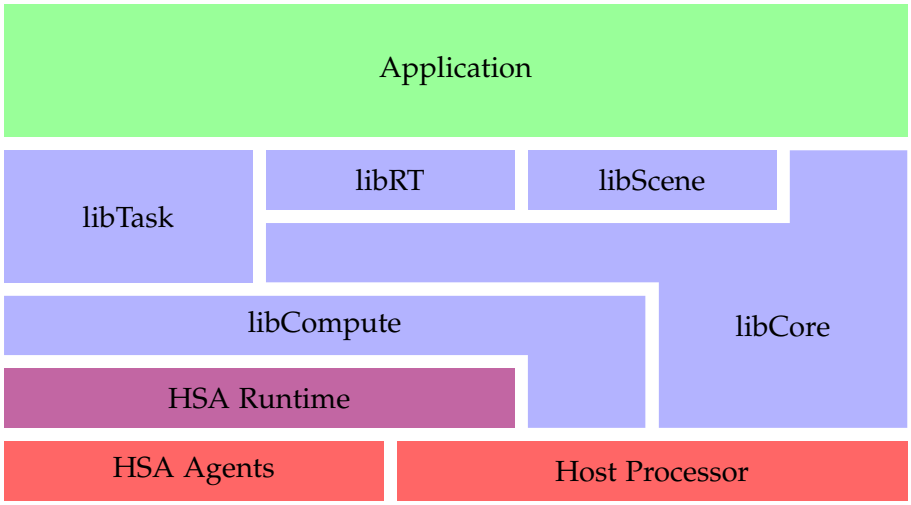



**Figure 6-1:** Organisation of RTKit Libraries

runtime library to provide agent discovery, memory allocators, kernel dispatch and synchronization primitives. Additionally, it provides related functionality for CPU execution, such as thread pools.

### 6.2.2 libCore - Fundamental Geometry Primitives

The core library, libCore, provides the basic geometric and mathematical primitives for implementing computer graphics algorithms.

Central to this library are C++ templates encapsulating fixed-length vectors, parameterized on both primitive data type and vector width. This type is analogous to hardware vector primitives, such as a Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX) vector, rather than a geometric vector. These vectors support the same set of C++ operators as the primitive data type corresponding to a vector element, along with a limited set of commonly used mathematical functions. Listing 6-1 provides an example of the use of this vector type. We also support a scalar form of this type, with an identical interface in order to aid us later in composing scalar and vector forms of more complex compound data structures.

Based upon our vector types, we provide a library of basic geometric primitives and functions. This includes representations of geometric points and vectors of various dimensionalities, along with lines, planes, triangles, spheres and bounding boxes. We

```
// Define aliases for 4 and 8-element SIMD vectors.
using float4 = rt::simd<float, 4>;
using float8 = rt::simd<float, 8>;

// Perform a series of component-wise arithmetic operations.
float8 a, b;
auto c = rt::sqrt(a) + 2.0f * b;
```

**Listing 6-1:** SIMD Vector Primitives in RTKit

also provide support for fundamental mathematical operators such as dot and cross products, and small matrix operations.

```
// Geometric primitives are expressed in terms of our rt::simd class.
// We define an alias for N points in 3D space, arranged such that
// component values are densely packed (x1,x2,x3,...,y1,y2,y3,...)
template<size_t N>
using point3f = rt::point<rt::simd<float, N>, 3>;

// 8 axis-aligned bounding boxes, arranged in structure of arrays
// form. Components are densely packed as per the point3f example.
rt::aabb<float, 8> soa_aabbs;

// 8 axis-aligned bounding boxes, arranged in array of structures
// form. Components are strided at 24-byte strides.
rt::aabb<float, 1> aos_aabbs[8];
```

**Listing 6-2:** Vectorized Geometric Primitives in RTKit

In combination with the Offload compiler described in the preceding chapter, this provides us with a library of basic computer graphics primitives, with compile-time configurable vector widths and elementary data types. Due to the Offload compiler, these primitives are usable on both the host processor and HSA kernel agents without modification.

This allows us to rapidly explore the impact of various vectorization strategies; or to evaluate the impact of transforming between array-of-structures and structure-of-arrays or packetized data layouts. An example of this technique is illustrated in listing 6-2.

For example, a widely used approach to accelerating coherent primary rays is to form rays into packets corresponding to the SIMD-width of the target processor. Several rays can then be tested in parallel against each node of an acceleration tree (Wald, Slusallek et al., 2001). This performs well for coherent rays, but loses efficiency for

secondary rays which exhibit a lower degree of coherency. An alternative approach is to construct a *multi bounding volume hierarchy*, a Bounding Volume Hierarchy (BVH) tree with a higher branching factor such that a single ray can instead be tested against multiple bounding boxes or primitives (Dammertz, Hanika and Keller, 2008; Ernst and Greiner, 2008; Viitanen et al., 2016; Wald, Benthin and Boulos, 2008)

By expressing all of our fundamental graphics primitives in terms of templated vectors, we can rapidly transform between these two implementations.

### 6.2.3 libRT - Intersection Tests and Acceleration Structures

Building upon our core library, our ray tracing library (libRT) provides tools for ray-geometry intersection testing and for the construction of BVH trees. We provide both traditional stack-based BVH traversal algorithms, and implementations of stackless traversal methods previously described by Laine (2010), Barringer and Akenine-Möller (2013) and Hapala et al. (2011). As with libCore, these traversal algorithms are usable on both host processors and kernel agents.

These traversal algorithms differ in their requirements for representing BVH tree nodes. Therefore, we additionally provide algorithms to construct BVH representations suitable for these different traversal algorithms. However, high-performance tree construction is not the primary focus of this work, and so we provide only multi-core CPU implementations on these construction algorithms. Other authors have addressed the parallel construction of BVH trees (Doyle, Fowler and Manzke, 2012; Lauterbach et al., 2009; Wald, 2007), and this might provide an avenue for future evaluation of our programming model and the mapping of work within heterogeneous systems.

### 6.2.4 libScene - Scene Graph and High Level Abstractions

The scene management library, libScene, provides abstractions of high-level graphics concepts such as cameras, meshes and lights. This is primarily a utility library handling the loading of scene and texture data, and managing a simple scene graph.

Listing 6-4 illustrates the loading of a scene from a file on disk, creation of a camera and generation of a packet of 64 primary rays from a set of two-dimensional sample points on the cameras focal plane. This can be coupled with the BVH construction

```cpp
// Construct a BVH tree from a scene.
rt::bvh_builder builder;
auto bvh = builder.build(scene);

// Create a packet of 8 rays.
rt::ray_packet<8> rays = ...;

// Construct a stack-based nearest-hit intersector.
rt::intersector<decltype(bvh),
                rt::nearest, rt::stack> intersect;

// Construct vectors to receive the intersected primitive id, ray
// distance and barycentric coordinates.
rt::simd<float, 8> t, u, v;
rt::simd<int32_t, 8> prim_ids;

// Test packet of 8 rays for ray-scene intersection.
intersect(rays, prim_ids, t, u, v);
```

**Listing 6-3**: BVH Construction and Ray-Scene Intersection Testing in RTKit

```cpp
// Load a scene from file. This loads geometry but does not construct
// acceleration structures.
rt::scene scene;
scene.load("hairball.obj");

// Create a camera and attach it to the scene.
rt::camera camera{...};
scene.add_camera(camera);

// Generate a set of 2D sample points on the cameras focal plane,
// then use the camera to generate a set of rays that can be tested
// against a BVH tree, as per previous examples.
rt::ray_packet<64> rays;
rt::point2f<64> samples = ...;
camera.generate_rays(rays, samples);
```

**Listing 6-4**: Scene Loading and Ray Generation in RTKit

and ray intersection example shown in listing 6-3 to perform a ray-scene intersection test.

### 6.2.5 libTask - Task Scheduling

The core and ray tracing libraries provide us with the tools to author both HSA kernel functions and host functions which implement the component tasks of a ray tracing pipeline. They also provide us with the primitives to explore the performance implications of low-level details such as vectorization and data layouts. However, they do not directly address the larger scale problem of mapping tasks to specific accelerator devices.

The tasking library (libTask) is the final component library in RTKit. This library provides primitives for controlling the scheduling and execution of work through use of task graphs; and an execution engine to consume task schedules, decompose them into smaller subtasks, and distribute them to CPU worker threads and HSA kernel agents.

Execution schedules are constructed as a graph of tasks, with nodes representing computational work and edges representing dependencies between tasks. The nodes perform no work directly. Instead, our execution engine will spawn a number of finer grained *task executors* for each task when the execution schedule is evaluated.

This is done to ensure that there is sufficient work available to keep multiple PUs active concurrently. We can illustrate this issue by means of an example. Consider a simple pipeline composed of two tasks assigned to separate PUs: a ray generation task, and a dependent ray-surface intersection testing task. If we intend to utilize this pipeline to evaluate a large corpus of rays, then there are three approaches to scheduling the processing of those rays that we might consider: per-ray, per-task, or batched.

We could evaluate individual rays and stream them between tasks and corresponding PUs one at a time, but this approach incurs significant communication overheads. Conversely, we could execute the ray generation task for all rays, and only begin executing the dependent ray-surface intersection task once all rays have been generated. This reduces communication costs, but leaves PUs idle waiting for preceding tasks to complete. Instead we adopt a batched approach, where multiple *task executors* each evaluate a subset of the rays. This enables dependent tasks to begin processing as

soon a subset of the output of preceding tasks becomes available. These batch sizes are user-configurable within RTKit. However, we find batches of 64 K rays to provide a reasonable default which provides effective utilization of the integrated GPU in our evaluation system.

Our execution engine is based one or more *work queues*, each with an associated pool of worker threads. Each *work queue* contains two lists of *task executors*: a *pending* list and an *active* list. The worker threads consume *task executors* from the lists, and either execute them directly, or dispatch them to kernel agents.

When an execution schedule is submitted to a *work queue*, our execution engine inspects the tasks that schedule is composed of, and identifies any tasks for which the dependencies are already satisfied. These are typically the producer tasks at the root of the graph. *Task executors* are then spawned for these tasks and added to the *pending* list.

The worker threads pull *executors* from the *pending* list, perform any necessary setup work such as memory allocation and then add the *executor* to the *active* list. At this stage the workers threads take one of two approaches. For CPU-backed tasks the worker executes the main computational work of the *executor*, and then marks the *executor* as complete. For tasks requiring execution on an HSA kernel agent, the worker thread dispatches the corresponding kernel to the HSA agent and immediately returns to polling the *active* and *pending* lists for further work to perform. In this case, the kernel agent itself will mark an *executor* as complete after the kernel has executed.

The worker threads additionally poll the *active* list for *executors* that have been marked complete, perform any necessary clean up work for these tasks, and then spawn a new set of *executors* for any dependent tasks. Priority is given to pulling completed *executors* from the *active* list, over beginning processing of a new *executor* from the *pending* list. This is done to limit possible starvation of PUs processing tasks located deeper in the execution schedule. Algorithm 1 provides further illustration of this process.

By structuring our task dispatch process in this manner we are able to ensure that sufficient work can be distributed to prevent starvation of PUs; that worker threads do no stall waiting for kernel completion; and that cyclic graphs or conditional dependencies can be supported.

```
while worker not terminating do
    poll active list;
    if completed executor available then
        pop executor from active list;
        perform executor teardown;
        spawn dependent executors;
        push dependent executors to pending list;
        continue
    end
    poll pending list;
    if pending executor available then
        pop executor from pending list;
        perform executor setup;
        push executor to active list;
        if is kernel executor then
            dispatch kernel;
        else
            perform executor body;
            mark executor complete;
        end
        continue
    else
        sleep until work available;
    end
end
```

**Algorithm 1:** Execution Engine Worker Thread Behaviour in RTKit

## 6.3 EVALUATION

In this section, we will explore the use of RTKit through a series of examples. We will begin with a simple ray casting example, and progress towards more complex examples.

### 6.3.1 Evaluation Objectives

In introducing this chapter, we defined three research questions:

- Is our programming model sufficient to implement a ray tracing pipeline?
- Can such ray tracing pipelines be implemented in a heterogeneous manner?
- Does doing so provide a performance benefit?

The first two of these objectives can be addressed by example. We have successfully utilized the Offload compiler and the associated runtime to implement a toolkit for evaluating ray tracing performance on heterogeneous systems. All of the benchmarks presented throughout section 6.3 are implemented using our C++ programming model for HSA, and all feature the use of both CPU and GPU cores.

The final research question addresses the core design goal for RTKit. Ray tracing performance can impacted by a wide variety of factors, including (but not limited to) scene dependent properties, hardware properties of the PU chosen to accelerate work, and algorithmic choices such as vectorization strategies or the specific acceleration tree traversal algorithm used.

All of these potential permutations make it impossible to provide a simple yes/no answer to this final research question, especially in the context of hypothetical future hardware. Instead RTKit aims to provide a framework to explore the complex interactions of these various permutations. We will provide ray-throughput measurements for a range of combinations of CPU and GPU task mappings, scalarized and packetized implementations, coherent and incoherent ray cohorts, nearest-point vs. occlusion intersection tests, and a variety of ray traversal algorithms. In doing so, we aim to provide further motivation for RTKit itself, and to provide some insights as to how these properties impact performance on our specific evaluation hardware.

### 6.3.2 Evaluation Plan

Throughout the course of this evaluation we will measure ray-throughput of a number of different ray tracing pipelines constructed using RTKit. These pipelines are each designed to highlight a specific characteristic or property. The throughput of these pipelines will be evaluated against six different evaluation scenes, which have varying geometric properties. We also provide measurements for scalarized and packet-based pipeline implementations utilizing both CPU and GPU cores.

In the preceding chapters, we have presented performance results generated by repeated sampling of isolated regions of code, with each sample typically representing a single GPU kernel, or time required to process a single image. This approach is less appropriate when attempting to benchmark the performance of RTKit. RTKit aims to achieve peak throughput by saturating all of the available PUs with work. As we described in section 6.2.5, RTKit relies upon a task scheduler to dispatch work to multiple PU in parallel. This requires a large body of available work to saturate a heterogeneous system and to reach a stable state.

Additionally, whilst RTKit is capable of generating per-task timestamps, multiple tasks will be processed concurrently. This leads to misleading throughput results if these timestamps are used directly to compute mean execution durations. Therefore, instead of attempting to measure latency for a single ray, we measure throughput over a large corpus of rays.

All of the results presented below are generated by processing a corpus of one billion rays per scene/PU/packet size exemplar. The RTKit runtime subdivides this corpus into batches of 64 K rays, resulting in an initial population of 16 K tasks. Each of these initial tasks will spawn further dependent tasks, based on the task graph.of the specific experiment. We derive ray-throughput by using the timestamps generated when the first task begins processing and the final task completes processing to compute a wall-clock duration.

This measurement strategy also results in the exclusion of the one-off cost of constructing acceleration data structures, which takes place before the first ray is traced.

### 6.3.3 Experimental Setup

As with preceding chapters, the benchmark system used in this evaluation is based around an AMD Kaveri APU, with CPU and GPU cores integrated into the same die and sharing the same DRAM. The processor is an AMD A10-7850K APU, containing four CPU cores and eight GPU compute units. Further details of these processor cores, including comparative theoretical instruction throughput, can be found in table 6-1.

| | **Central Processing Unit** | **Graphics Processing Unit** |
|---|---|---|
| Cores / Compute Units | 4 (2 shared FPUs) | 8 |
| Clock Speed | 3700 MHz | 720 MHz |
| Instructions / Clock | 16 (8-wide SIMD FMA) | 128 (4 x 16-wide SIMD FMA) |
| Theoretical Peak FLOPS | 118.40 GFLOPS | 737.20 GFLOPS |

**Table 6-1:** Advanced Micro Devices A10-7850K Accelerated Processing Unit

We make use of six different evaluation scenes. These scenes have differing triangle counts and geometric structure, and consequently we will observe significant performance differences between these scenes. Table 6-2 provides details of these scenes, along with example images. These are generated using either path tracing or ambient occlusion pipelines implemented with RTKit.

In describing these scenes in table 6-2, we introduce the concept of BVH traversal complexity. In order to resolve whether a ray intersects with a surface within a scene, we must perform a depth-first search of the BVH tree, testing the ray for intersection with a series of nested AABBs that encompass the internal nodes of the BVH tree. This process continues until a leaf node is reached, where a ray is tested for intersection with the surface itself. If the ray fails to intersect with a surface, then we must backtrack through the tree and explore the remaining unvisited branches.

The performance of ray-scene intersection tests is therefore strongly influenced by the number of internal nodes that must be traversed and ray-AABB intersection tests that must be performed. It follows that this traversal complexity is a property of the virtual scene. Scenes with large regions of space containing little or no geometry, such as the Dragon and Lucy scenes can be traversed in fewer steps, whilst scenes with complex overlapping geometry, such as the Hairball scene are more challenging. Section 6.3.5 provides both an example of the use of RTKit to visualize this BVH traversal complexity, and the visualization itself.

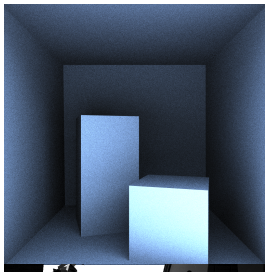
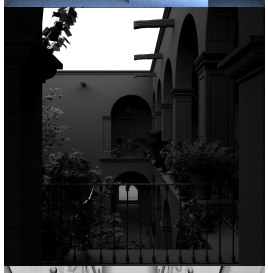
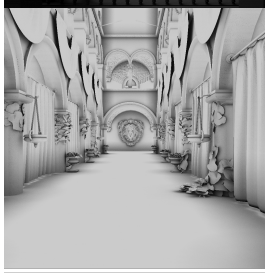
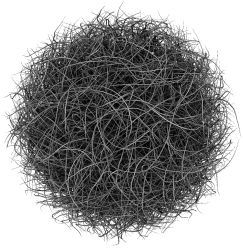
| | | |
|---|---|---|
|  | Cornell Box<br>140 K triangles | The classic Cornell Box scene, recursively subdivided to increase the polygon count. This scene has low BVH traversal complexity for both the mean and worst-case scenarios. |
|  | San Miguel<br>6.60 M triangles | A high-polygon architectural scene, with the highest mean BVH traversal complexity. |
|  | Crytek Sponza<br>260 K triangles | An architectural scene with a wide range of polygon sizes and medium BVH traversal complexity. |
|  | Hairball<br>2.90 M triangles | The Hairball scene has particularly complex structure, with large numbers of small elongated triangles. This results in large numbers of overlapping AABBs within the BVH tree. Equivalent mean BVH traversal complexity to Sponza, but extremely high worst-case complexity. |
|  | Stanford Dragon<br>870 K triangles | Generated from a 3D scan of a statuette, this scene consists of small, regularly sized polygons. It has a low mean BVH traversal complexity due to large regions of empty space, and comparable worst-case complexity to Sponza. |
|  | Lucy<br>28 M triangles | A high-polygon 3D scan of a statue. This scene combines an extremely high polygon count with low mean BVH complexity. |

**Table 6-2:** Evaluation Scenes for RTKit

### 6.3.4 Coherent Ray Casting

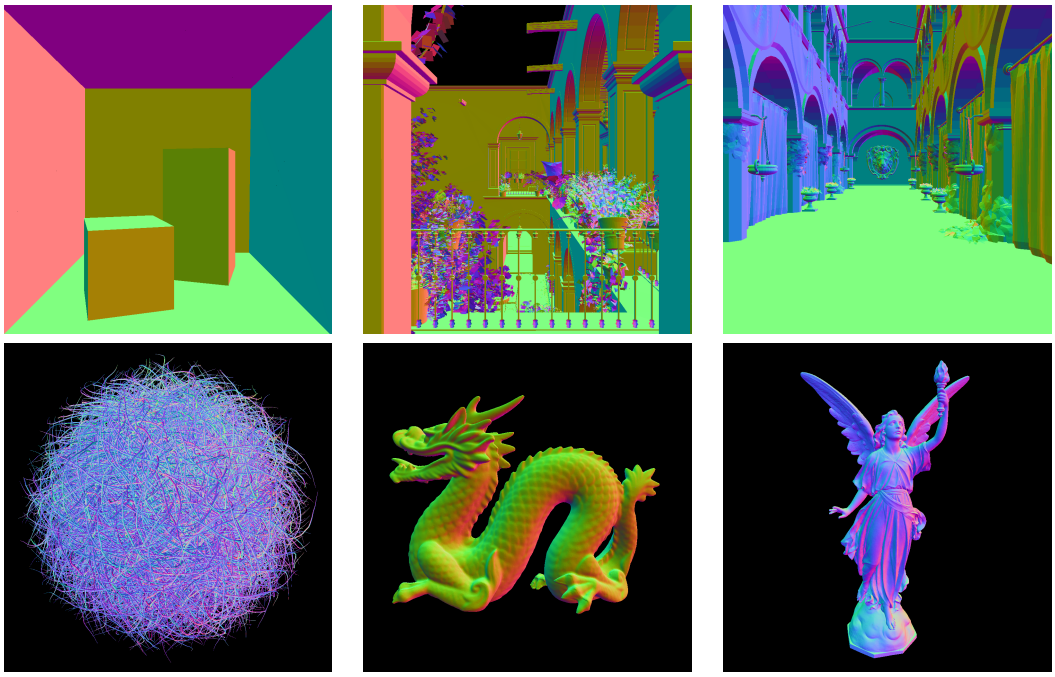


**Figure 6-2:** Ray Casting Examples - Nearest Ray-Surface Intersection

We begin our evaluation with a simple ray casting example. Ray casting can be viewed as the simplest form of ray tracing algorithm, consisting only of a primary ray step, with no secondary bounces.

We will measure ray throughput for scalar and packetized ray traversal algorithms, using both the CPU and GPU cores. This approach allows us to verify that despite sharing a common memory subsystem, the correct choice of processor core and vectorization strategy can result in significant performance gains.

For this benchmark, we decompose ray casting down into five stages. Firstly, we generate sample points on the image plane. We then generate outgoing rays by tracing a line that intersects both the focal point of the virtual camera and the sample point on the image plane. These rays are then tested for intersection with the scene geometry. We can now calculate an output colour for each sample point, based on whether the rays intersected the scene geometry. Finally, the output colours can be accumulated into the output image. We can see this simple pipeline illustrated in figure 6-3.

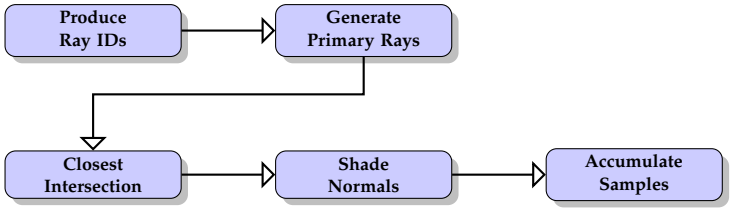



**Figure 6-3:** Task Graph for Ray Casting - Nearest Ray-Surface Intersection

For our first ray casting benchmark, we will compute the location of the ray-surface intersection point closest to the camera on each ray. For visualization purposes, we colour each ray-surface intersection based on the surface normal vector at the point of intersection. We can see examples of this visualization in figure 6-2, and an example implementation of this pipeline in listing 6-5.

We generate several task graphs, each consisting solely of tasks executed on either the CPU or GPU, and with varying packet widths. The only exception to this is the ray identifier production step, which is executed on the CPU in all cases.

This variation of packet widths provides the first practical example of RTKit enabling us to manipulate vectorization and data layouts. The scalar examples result in rays, intersection and shading data arranged in array-of-structures layouts, while the packetized layouts result in data structures with packed layouts corresponding to the packet width.

Figure 6-4 and table 6-3 illustrate effective ray throughput for the complete graph. From this graph, we can immediately draw a number of conclusions. Pipelines evaluated solely on the CPU see some benefit from vectorization, but the observed speedup factor is significantly less than the SIMD width. Several factors are responsible for this.

| Scene | Cornell Box | San Miguel | Sponza | Hairball | Dragon | Lucy |
|---|---|---|---|---|---|---|
| CPU (Single Ray) | 11.51 | 5.47 | 5.88 | 6.67 | 14.63 | 16.18 |
| CPU (Packet x 4) | 14.62 | 5.84 | 7.00 | 7.02 | 18.43 | 20.41 |
| CPU (Packet x 8) | 15.68 | 6.25 | 7.21 | 7.56 | 20.04 | 22.26 |
| GPU (Single Ray) | 3.82 | 1.07 | 1.58 | 1.69 | 5.61 | 6.98 |
| GPU (Packet x 64) | 29.40 | 8.62 | 12.86 | 10.23 | 37.61 | 41.67 |

**Table 6-3:** Coherent Ray Casting Throughput - Nearest Ray-Surface Intersection (Mrays/s)

Firstly, whilst the CPU in our evaluation system supports 8-wide SIMD through the AVX instruction set, this is accomplished by executing a pair of 4-wide SIMD

```
// Define type aliases for the task nodes. Modifying the template
// parameters allows us to retarget particular nodes at different
// HSA agents, or experiment with different packet/vector widths.
using ray_id_producer_t =
    rt::ray_id_producer_t<1>;
using primary_ray_producer_t =
    rt::primary_ray_producer_t<N, target::cpu>;
using nearest_intersector_t =
    rt::nearest_hit_t<N, target::cpu>;
using shade_normals_t =
    rt::false_colour_geometric_normals_t<N, target::cpu>;
using accumulator_t =
    rt::accumulate_radiance_t<N, target::cpu>;

// Construct the task graph, starting with a ray ID generator.
auto ids  = make_task<ray_id_producer_t>();

// Construct a graph node to map ray IDs to sample points on
// the camera image plane.
auto rays = make_task<primary_ray_producer_t>(ids, camera,
                                              fb.width(), fb.height(),
                                              samples_per_pixel,
                                              nullptr, nullptr);

// Construct a graph node to compute ray-geometry intersections.
auto intersections = make_task<nearest_intersector_t>(rays, scene);

// Construct a graph node to shade intersections, in this case in
// false colour based on the surface normal vector.
auto shading = make_task<shade_normals_t>(intersections,
                                          scene.triangles(),
                                          scene.vertex_positions());

// Construct a graph node to accumulate radiance into the final
// image.
auto accumulator = make_task<accumulator_t>(shading, &fb,
                                              samples_per_pixel);
```

**Listing 6-5:** Nearest Ray-Surface Intersection Ray Casting

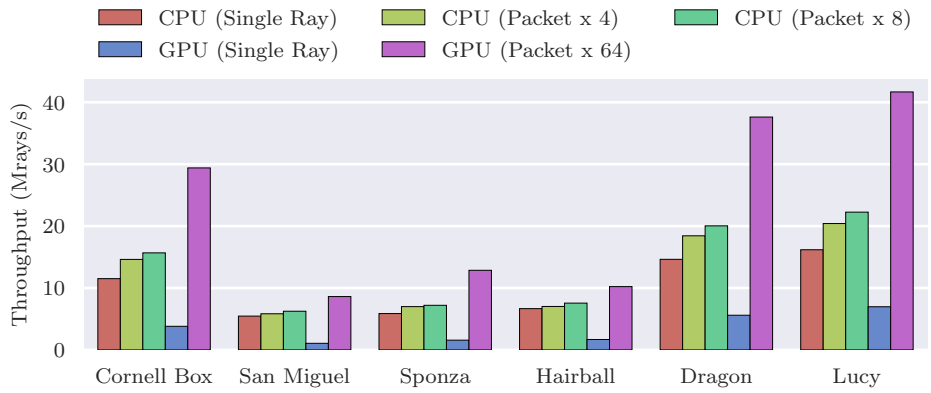



**Figure 6-4:** Coherent Ray Casting Throughput - Nearest Ray-Surface Intersection

operations. As such AVX instructions offer little benefit on our evaluation platform. Secondly, not all stages of the pipeline are well suited to SIMD parallelisation. Most notably, the final accumulation stage is effectively a scatter operation, with minimal arithmetic. Neither the SSE nor AVX instruction sets provide scatter instructions, and so this stage is always implemented in a scalar form. Finally, some of the stages in our pipeline are amenable to compiler auto-vectorization, even when expressed in a scalar form. In this case, the scalar implementation is implicitly benefiting from vectorization and explicit vectorization offers no additional benefit.

For the GPU, we evaluate scalar and 64-wide packetized GPU implementations, with 64 corresponding to the wavefront width of the integrated GPU in our evaluation system. Here we see significant differences in performance based on packet size.

The 64-wide packetized GPU implementation offers a speedup of up to 1.87× over the best CPU implementation. By contrast, the scalar implementation performs poorly. With the exception of data layouts, these two implementations are algorithmically identical. In both the scalar and packetized implementations, each work-item evaluates a single scalar ray, with the packetized implementation performing additional work to unpack and pack data. For the scalar GPU kernel, the input rays are laid out according to an array-of-structures layout. As a result, a pair of neighbouring work-items each loading the same component value from their corresponding input ray will perform reads separated by a stride equivalent to the size of a scalar ray (32 bytes). By contrast, neighbouring work-items loading the same component value in the packetized form load at 4 byte strides.

This is consistent with our investigation into HSAIL vector performance in section 5.4.4, and particularly figure 5-2, where we observed significant performance differences between densely packed and strided stores.

It is important to note that these results represent a best-case scenario for the GPU implementations. This benchmark is solely limited to primary rays. As a result, neighbouring rays within each batch are highly coherent, resulting in low branch divergence and a high proportion of coalesced memory accesses.

Whilst the use of the packetized GPU implementation provides a net benefit for all benchmarks, we note that the total speedup is less pronounced for the Hairball (1.35×) and San Miguel (1.38×) scenes. These scenes are noteworthy for having regions of high BVH traversal complexity, where a ray must traverse many overlapping BVH nodes before reaching a final intersection point. By contrast, whilst the Lucy scene has an extremely high triangle count, a lower proportion the BVH nodes overlap. Additionally, large portions of the Lucy scene are entirely empty, allowing the BVH traversal kernel to exit rapidly.

### 6.3.5 Visualizing BVH Traversal Complexity

In examining our ray-throughput results in section 6.3.4, we noted that on certain scenes the packetized GPU implementation produced a smaller speedup than on other scenes, and we attributed this to a higher BVH traversal complexity for these scenes.We can visualize this within RTKit by replacing the intersection test graph node with a new one that counts the number of BVH tree nodes visited by each ray during the tree traversal. Figure 6-5 and table 6-4 illustrate the results of this visualization.

| Scene | Mean | Maximum |
|---|---|---|
| Cornell Box | 18.32 | 100.00 |
| San Miguel | 82.99 | 474.00 |
| Sponza | 53.44 | 159.00 |
| Hairball | 59.25 | 1579.00 |
| Dragon | 11.60 | 160.00 |
| Lucy | 8.20 | 276.00 |

**Table 6-4:** BVH Nodes Visited Per Ray During Nearest Ray-Surface Intersection Test

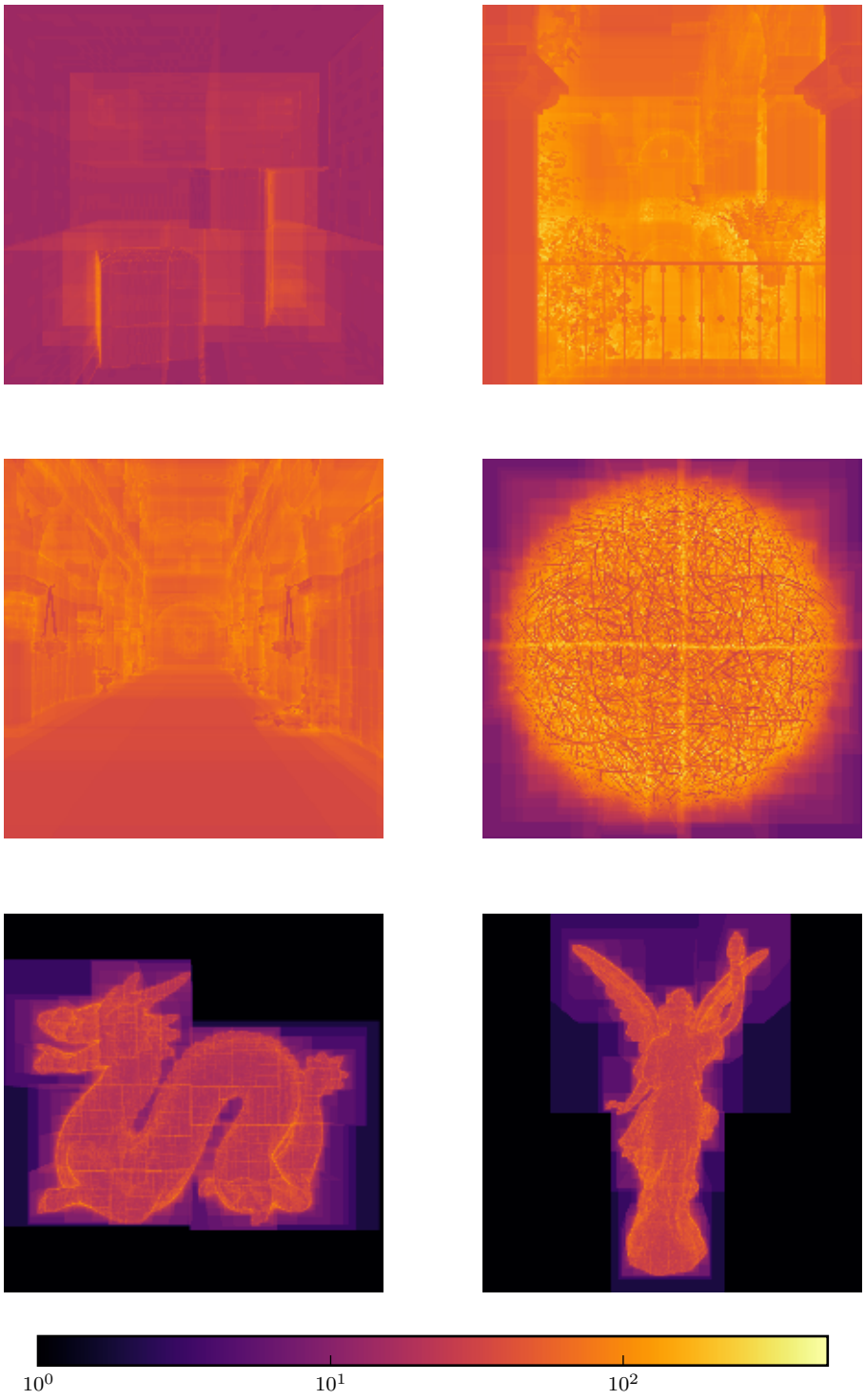



**Figure 6-5:** Visualization of Number of BVH Nodes Visited During Nearest Ray-Surface Intersection Test

```
// Construct generate rays as per previous example.
auto ids  = make_task<ray_id_producer_t>();
auto rays = make_task<primary_ray_producer_t>(ids, ...);

// The ray-surface intersction node is replaced with a node to count
// BVH nodes traversed while computing ray-geometry intersections.
auto node_counts = make_task<node_counting_intersector_t>(rays, ...);

// Replace the shading and accumulation node with direct scalar
// accumulation.
auto accumulator = make_task<scalar_accumulator_t>(node_counts, ...);
```

**Listing 6-6:** Modified Ray Casting Pipeline for Visualizing BVH Tree Traversal Complexity in RTKit

Listing 6-6 illustrates how this can be implemented using RTKit. The ray-surface intersection test node shown in listing 6-5 is replaced with an alternative intersection node, which returns the number of BVH nodes traversed per ray rather than returning a ray-surface intersection point. Additionally, the shading and accumulation stages are replaced with direct accumulation of this scalar value into the frame buffer.

### 6.3.6 Agent Utilization

As we discussed in section 6.2.5, RTKit relies upon a task graph and scheduler to schedule work between processor cores and to manage dependencies between tasks. This task management and scheduling has a non-zero cost, and it is important to verify that the scheduler itself does not act as a bottleneck, or consume a significant proportion of available computation resources. We can make use of our coherent ray casting benchmark to examine the efficiency of our task scheduler and the overheads imposed by RTKit.

RTKit allows us to track per-task timestamps at various points during task scheduling and execution. This allows us to calculate the time spent executing tasks, as distinct from time spent on task scheduling, or idling waiting for work to become available. In cases where scheduling overhead is low, we should expect to observe close to 100% utilization for the PU responsible for task execution, whilst lower utilization would potentially indicate either excessive resource consumption by the scheduler itself, or stalls due to dependencies between tasks limiting the quantity of available

work to process. Table 6-5 provides a summary of PU utilization for task processing, measured over the coherent ray casting benchmark presented in section 6.3.4.

| Scene | CPU Utilization | GPU Utilization |
|---|---|---|
| CPU (Single Ray) | 0.99 | 0.00 |
| CPU (Packet x 4) | 0.98 | 0.00 |
| CPU (Packet x 8) | 0.97 | 0.00 |
| GPU (Single Ray) | 0.01 | 1.38 |
| GPU (Packet x 64) | 0.05 | 1.28 |

**Table 6-5:** Agent Utilization During Coherent Ray Casting Throughput

From table 6-5, we can draw several conclusions. Firstly, for CPU pipelines, 97-99% of processing time is spent on executing tasks from our task graph, across all four cores of our CPU. This figure does not address the efficiency of implementation of the tasks themselves, but instead suggests that the overhead of our task scheduler does not significantly impact performance.

Secondly, when executing the GPU pipelines we still consume a small amount of host processor time on task execution. This can be attributed to a combination of the ray identifier production step; additional host-side processing prior to kernel dispatch, such as the allocation of memory for task outputs; and kernel dispatch and synchronization costs. These factors cumulatively only account for 1-5% of CPU time.

Surprisingly, our reported GPU utilization exceeds 100%. We record time stamps when each kernel dispatch is added to the user-mode queue for the GPU agent, along with when each kernel dispatch begins and completes execution. The figures shown in table 6-5 are derived from the kernel execution timestamps, and so are indicative of the concurrent execution of multiple kernels on our GPU.

This concurrent kernel execution is in part a consequence of our decision to subdivide rays into batches, as described in section 6.2.5. Without this subdivision subsequent tasks in our pipeline, such as shading, would be unable to make forward progress until all rays had cleared the preceding stages. Conversely, excessive subdivision results in increased communication overhead, as large numbers of relatively small tasks must then be managed by our scheduler and the GPU queues. We use a default batch size of 64 K rays, which we have found empirically to provide a good baseline.

### 6.3.7 Occlusion Testing

**Figure 6-6:** Ray Casting Examples - Any Ray-Surface Intersection

Thus far we have explored the performance of ray casting algorithms which attempt to find the closest surface intersection point located on a ray. However, for some use-cases it is sufficient to identify the existence of any intersection point along a ray segment, without regard to the precise location. For example, these tests can be used to determine whether a light source is occluded. Whilst these occlusion tests can be resolved by using the same closest point intersection test, a specialized any hit test has opportunities to perform early exits and to relax traversal ordering constraints to improve coherence. Figure 6-6 shows silhouettes of the Hairball, Stanford Dragon and Lucy scenes generated using a pipeline modified to simply identify the existence of any intersection point. We exclude the Cornell Box, Sponza and San Miguel scenes here purely because the geometry in these scenes encompasses the whole image, resulting in visually uninteresting silhouettes.

Compared to the closest intersection tests, occlusion tests potentially allow for an earlier exit from the traversal of acceleration data structures. In this benchmark, we will measure ray throughput when using an occlusion test instead of a closest hit test. This is evaluated over the same combination of test scenes, PU mappings and packet sizes as the coherent ray casting benchmark presented in section 6.3.4.

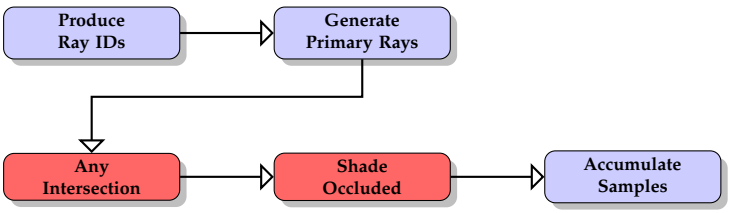



**Figure 6-7:** Task Graph for Ray Casting - Any Ray-Surface Intersection

Modifying the task graph from figure 6-3 to produce a new graph implementing an occlusion test algorithm simply requires replacing two nodes. Firstly, we must replace the actual intersection test node with a new node that exits early on finding any intersection. Secondly, as the new intersection node only calculates the existence or non-existence of an intersection point, and not the specific geometric primitive intersected, we must also replace the shading node. In this example, we simply provide a node that shades samples corresponding to occluded rays white. The new graph with the modified nodes highlighted is shown in figure 6-7, while listing 6-7 provides an illustration of the required code changes in RTKit. This serves as an illustration of the power of our approach. By replacing two lines of code, we are able to repurpose our task graph. Whilst in this instance the change was made for algorithmic reasons, the same approach can be used to move a node to a different PU, modify memory allocators or manage data movement and layout.

```
// Construct generate rays as per previous examples.
auto ids  = make_task<ray_id_producer_t>();
auto rays = make_task<primary_ray_producer_t>(permuted_ids, ...);

// Replace the nearest-surface intersection test with an any-surface
// test. This returns hit/miss status rather than an intersection
// point.
auto occlusions = make_task<any_intersector_t>(rays, ...);

// We cannot usefully render surface properties, with any-surface
// intersections so replace the shading stage with a simple
// silhouette.
auto shading = make_task<shade_occluded_t>(occlusions, ...);

// Continue the remainder of the pipeline as per the nearest hit
// benchmark.
auto accumulator = make_task<accumulator_t>(shading, ...);
```

**Listing 6-7**: Modified Ray Casting Pipeline for Any-Surface Intersection in RTKit

Figure 6-8 and table 6-6 illustrate the performance of our modified ray casting pipeline.

The any-surface intersection test provides opportunities for early exit from the BVH tree search. As a result, we would expect equivalent or improved performance relative to the nearest-surface intersection test results previously illustrated in figure 6-4 and table 6-3. Table 6-7 shows the relative speedup between the any-surface and nearest-surface intersection pipelines.

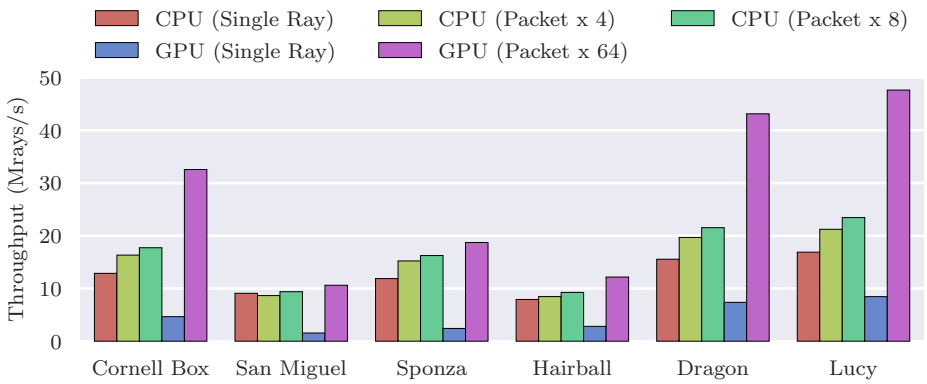



**Figure 6-8:** Coherent Ray Casting Throughput - Any Ray-Surface Intersection

| Scene | Cornell Box | San Miguel | Sponza | Hairball | Dragon | Lucy |
|---|---|---|---|---|---|---|
| CPU (Single Ray) | 12.87 | 9.09 | 11.88 | 7.92 | 15.56 | 16.90 |
| CPU (Packet x 4) | 16.33 | 8.66 | 15.23 | 8.47 | 19.69 | 21.23 |
| CPU (Packet x 8) | 17.74 | 9.39 | 16.25 | 9.26 | 21.54 | 23.45 |
| GPU (Single Ray) | 4.66 | 1.56 | 2.42 | 2.82 | 7.38 | 8.46 |
| GPU (Packet x 64) | 32.58 | 10.62 | 18.73 | 12.17 | 43.14 | 47.65 |

**Table 6-6:** Coherent Ray Casting Throughput - Any Ray-Surface Intersection (Mrays/s)

| Scene | Cornell Box | San Miguel | Sponza | Hairball | Dragon | Lucy |
|---|---|---|---|---|---|---|
| CPU (Single Ray) | 1.12 | 1.66 | 2.02 | 1.19 | 1.06 | 1.04 |
| CPU (Packet x 4) | 1.12 | 1.48 | 2.18 | 1.21 | 1.07 | 1.04 |
| CPU (Packet x 8) | 1.13 | 1.50 | 2.25 | 1.22 | 1.07 | 1.05 |
| GPU (Single Ray) | 1.22 | 1.46 | 1.53 | 1.67 | 1.32 | 1.21 |
| GPU (Packet x 64) | 1.11 | 1.23 | 1.46 | 1.19 | 1.15 | 1.14 |

**Table 6-7:** Coherent Ray Casting Speedup - Any vs. Nearest Ray-Surface Intersection

The smallest gains are observed in the Dragon and Lucy scenes. This is relatively unsurprising as these scenes exhibited relatively low quantities of BVH nodes traversed per ray when we explored traversal complexity in figure 6-5 and table 6-4. These scenes exhibit large regions of empty space, where there is little or no opportunity for an earlier exit from BVH traversal relative to the nearest-surface intersection kernel.

### 6.3.8 Incoherent Ray Casting

Ray-geometry intersection tests for primary rays can generally be expected to perform well on a GPU due to the high degree of similarity between neighbouring rays within a batch of rays. This property increases the likelihood that all of the rays mapped onto a single wavefront will traverse the BVH tree in the same order, and access the same memory locations.

However, this property is less likely to hold true for secondary rays. Techniques such as ambient occlusion and Monte-Carlo path tracing involve the random sampling of the visible hemisphere oriented around the surface normal vector at each intersection point. This random sampling results in a low degree of coherence or similarity in outgoing ray directions when evaluated across a set of rays.

Incoherent rays are known to be challenging for both GPUs and packet-based ray tracers due to a combination of introducing control-flow that is non-uniform across SIMD lanes, and irregular memory access patterns. In this benchmark, we aim to evaluate the performance impact of incoherent rays by measuring ray throughput over an artificially permuted batch of rays. We do this by using the same set of test scenes, algorithms and PU mappings as in the preceding benchmarks, but randomly shuffling the generated rays.

We can provide some insight into how this might be expected to impact performance simply by inserting a new node into our task graph. If we permute each ray identifier before ray generation, we can alter the pattern in which the focal plane of the virtual camera is sampled. In figure 6-9 and listing 6-8, we insert a new graph node to apply a hashing function to transform sequential ray identifiers into a pseudo-random distribution. This in turn has the effect of producing a pseudo-random distribution of rays across the camera frustum.

In this particular example we are transforming ray identifiers in order to artificially induce random sampling and reduce ray coherence. However, the same technique can also be applied to attempt to improve locality by mapping a linear sequence of ray identifiers into tiles or onto a space-filling Hilbert curve (Sagan, 2012).

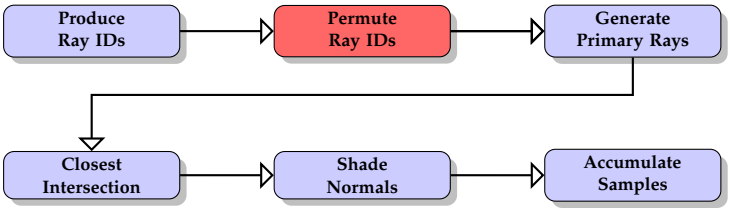



**Figure 6-9:** Task Graph for Ray Casting - Nearest Ray-Surface Intersection with Random Ray Shuffling

```
// Construct generate rays as per previous examples.
auto ids  = make_task<ray_id_producer_t>();

// Permute the ray ids with a hashing function.
auto permuted_ids  = make_task<wang_hash>();

// Continue the remainder of the pipeline as per the nearest hit
// benchmark.
auto rays = make_task<primary_ray_producer_t>(permuted_ids, ...);
auto intersections = make_task<nearest_intersector_t>(rays, ...);
auto shading = make_task<shade_normals_t>(intersections, ...);
auto accumulator = make_task<accumulator_t>(shading, ...);
```

**Listing 6-8:** Modified Ray Casting Pipeline for Simulating the Impact of Incoherent Rays in RTKit

The results presented thus far represented the best case scenario for both CPU and GPU implementations. Ray identifiers were generated sequentially, leading to the generation of rays with an extremely high degree of similarity between neighbouring rays in the ray buffers. This in turn leads to effective cache usage, low branch divergence and a high degree of memory coalescing between neighbouring SIMD lanes for both CPU and GPU implementations. This pattern is reasonable for primary rays emitted from a camera or light source. However, this high level of coherence is unlikely to maintained in all cases. Monte-Carlo integration techniques are commonly used for sampling incident light at a shading point. This results in reduced coherence for secondary rays.

Figure 6-10 and table 6-8 illustrate throughput for incoherent rays. These results use an identical corpus of rays, and identical test scenes and camera positions to the

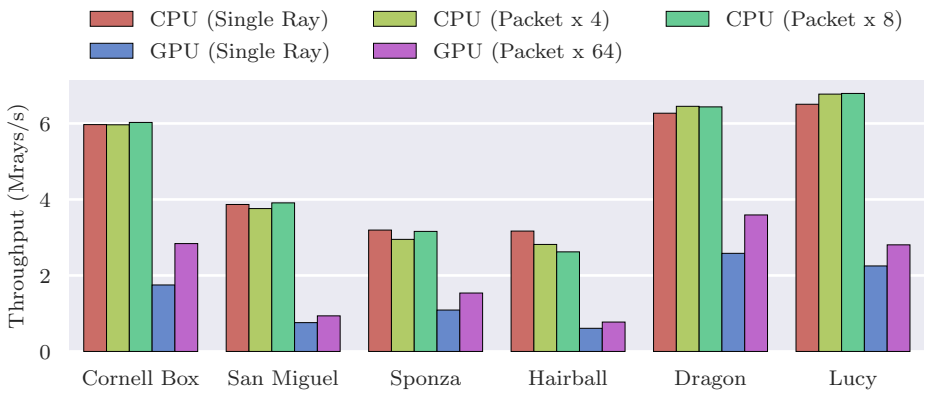



**Figure 6-10:** Incoherent Ray Casting Throughput - Nearest Ray-Surface Intersection

| Scene | Cornell Box | San Miguel | Sponza | Hairball | Dragon | Lucy |
|---|---|---|---|---|---|---|
| CPU (Single Ray) | 5.97 | 3.87 | 3.19 | 3.17 | 6.27 | 6.50 |
| CPU (Packet x 4) | 5.96 | 3.76 | 2.95 | 2.82 | 6.45 | 6.77 |
| CPU (Packet x 8) | 6.03 | 3.91 | 3.16 | 2.62 | 6.44 | 6.79 |
| GPU (Single Ray) | 1.75 | 0.76 | 1.09 | 0.61 | 2.58 | 2.25 |
| GPU (Packet x 64) | 2.84 | 0.94 | 1.54 | 0.77 | 3.59 | 2.81 |

**Table 6-8:** Incoherent Ray Casting Throughput - Nearest Ray-Surface Intersection (Mrays/s)

coherent results shown in figure 6-4 and table 6-3. The only difference between the two cases is that the rays used in the benchmarks shown in figure 6-10 and table 6-8 have effectively been shuffled into a pseudo-random order.

Whilst all five implementations show some degradation in performance when rays are sorted into an incoherent order, the performance degradation is particularly severe for both GPU implementations. Our results for coherent rays (figure 6-4 and table 6-3) showed the integrated GPU providing the highest throughput in all cases. By contrast, we find that the use of the integrated GPU provides a significant performance degradation relative to the CPU for incoherent rays (figure 6-10 and table 6-8). This occurs regardless of whether the GPU implementation is packetized.

| **Scene** | **Cornell Box** | **San Miguel** | **Sponza** | **Hairball** | **Dragon** | **Lucy** |
|---|---|---|---|---|---|---|
| CPU (Single Ray) | 1.93 | 1.41 | 1.84 | 2.11 | 2.33 | 2.49 |
| CPU (Packet x 4) | 2.45 | 1.55 | 2.37 | 2.49 | 2.86 | 3.01 |
| CPU (Packet x 8) | 2.60 | 1.60 | 2.28 | 2.89 | 3.11 | 3.28 |
| GPU (Single Ray) | 2.18 | 1.41 | 1.45 | 2.77 | 2.17 | 3.10 |
| GPU (Packet x 64) | 10.36 | 9.20 | 8.37 | 13.22 | 10.47 | 14.85 |

**Table 6-9**: Ray Casting Speedup - Nearest Ray-Surface Intersection Coherent vs. Incoherent

These differences are further highlighted in table 6-9, which shows the relative speedup for coherent versus incoherent rays.

That incoherent rays lead to a performance degradation in GPU and SIMD packet ray tracers is relatively well understood. Given the severity of the performance degradation shown for the packet-based GPU implementation, some form of sorting to improve coherence may appear attractive. Indeed, previous work by Mansson, Munkberg and Akenine-Möller (2007) has explored this topic. In practice, whilst sorting can improve the performance of the ray-geometry intersection step in isolation, the additional cost of performing the sort appears to out weigh the performance gain. In particular, sorting imposes additional bandwidth and synchronization costs, particularly for GPUs which typically require the use of specialized sorting algorithms due to execution model constraints. Furthermore, the goal of maximizing per-packet coherence favours selecting from large ray corpora. However, increasing the size of the candidate ray population also increases sorting costs.

### 6.3.9 Fine and Coarse-Grained Memory

When we examined the performance of coherent ray casting in sections 6.3.4 and 6.3.7, we observed excellent performance for the packetized GPU implementation, but poor performance for the scalar GPU implementation. This held true for both the nearest-surface (figure 6-4, table 6-3) and any-surface (figure 6-8, table 6-6) intersection pipelines. This is despite the fact that the scalar and packetized kernels are algorithmically identical with the exception of data layouts.

The experiments shown in sections 6.3.4 and 6.3.7 use fine-grained memory allocations throughout. However, previous benchmarks described in chapter 5 showed significant performance discrepancies dependent on whether fine or coarse-grained memory allocations were used.

In this benchmark, we aim to establish whether the poor performance of the single-ray GPU implementations can be attributed to the choice of memory allocator, and consequently whether this is a significant parameter for developers to consider when attempting to choose the most appropriate combination of PU and packetization approach.

We can evaluate the potential impact of coherency granularity by varying the memory allocators used for our graph. RTKit allows us to configure the memory allocator individually for each node in our graph. In this example, we will evaluate pipeline throughput utilizing both coarse and fine-grained memory allocators throughout the pipeline. These benchmarks are only relevant to the GPU-based implementation, as it is the only agent within our evaluation system that exposes a coarse-grained memory region.

Figures 6-11 and 6-12 illustrate throughput for nearest-surface and any-surface intersection pipelines respectively, for pipelines based on both fine and coarse-grained allocations. Additionally, table 6-10 provides the combined performance results.

We have not included any results for the San Miguel scene, or nearest-surface intersection results for Hairball. This is due to a limitation in our scheduler. Coarse-grained memory is a limited resource on our evaluation system, and our scheduler does not currently track the quantity of coarse-grained memory that has been consumed prior to launching tasks. We can see from figure 6-4 and table 6-3 that San Miguel and Hairball demonstrated the lowest throughput in earlier benchmarks. As a result, these scenes result in the highest quantities of ray batches in flight concurrently and so exhaust the available coarse-grained memory. This could be resolved if our sched-

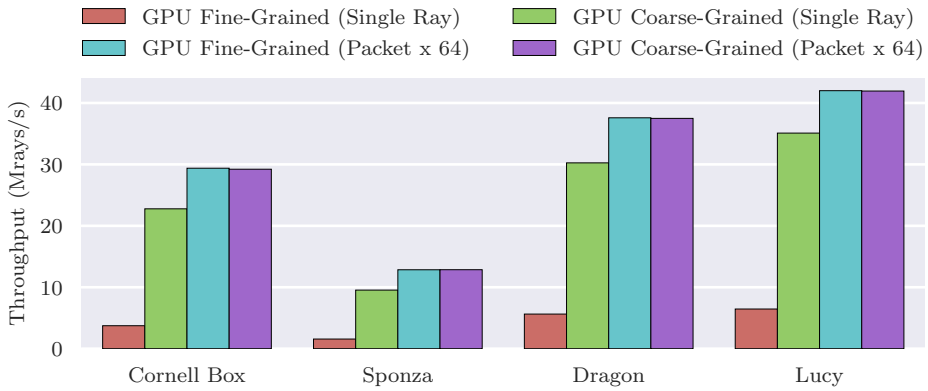



**Figure 6-11:** Coherent GPU Ray Casting Throughput - Coarse and Fine-Grained Memory - Nearest Ray-Surface Intersection

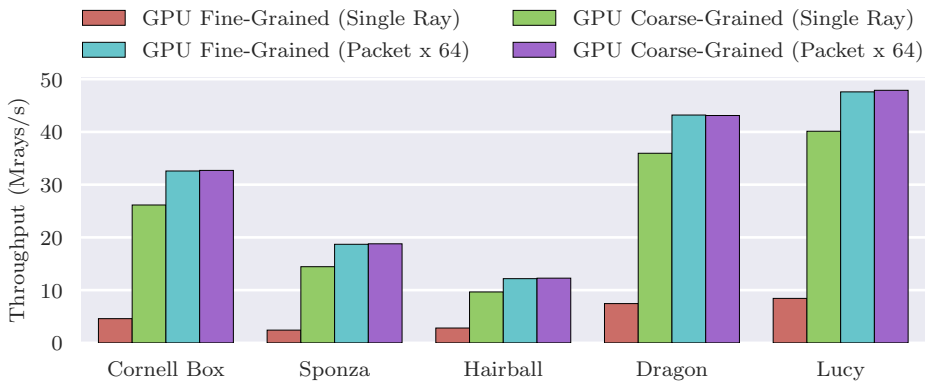



**Figure 6-12:** Coherent GPU Ray Casting Throughput - Coarse and Fine-Grained Memory - Any Ray-Surface Intersection

| Scene | Cornell Box | San Miguel | Sponza | Hairball | Dragon | Lucy |
|---|---|---|---|---|---|---|
| **GPU (Single Ray) - Nearest Ray-Surface Intersection** | | | | | | |
| Fine-Grained | 3.74 | | 1.57 | | 5.64 | 6.46 |
| Coarse-Grained | 22.77 | | 9.54 | | 30.25 | 35.09 |
| **GPU (Single Ray) - Any Ray-Surface Intersection** | | | | | | |
| Fine-Grained | 4.59 | | 2.42 | 2.81 | 7.45 | 8.43 |
| Coarse-Grained | 26.14 | | 14.44 | 9.66 | 35.95 | 40.13 |
| **GPU (Packet x 64) - Nearest Ray-Surface Intersection** | | | | | | |
| Fine-Grained | 29.38 | | 12.85 | | 37.58 | 42.00 |
| Coarse-Grained | 29.21 | | 12.86 | | 37.49 | 41.94 |
| **GPU (Packet x 64) - Any Ray-Surface Intersection** | | | | | | |
| Fine-Grained | 32.59 | | 18.69 | 12.18 | 43.20 | 47.59 |
| Coarse-Grained | 32.70 | | 18.78 | 12.27 | 43.11 | 47.89 |

**Table 6-10:** Coherent GPU Ray Casting Throughput - Fine-grained versus Coarse-grained Memory (Mrays/s)

uler tracked memory allocations and throttled the dispatch of tasks to avoid resource starvation. This remains future work.

From figure 6-11, figure 6-12 and table 6-10, we can make two observations. Firstly, the use of coarse-grained memory allocations appears to have a negligible impact on the packetized pipelines. Secondly, the use of coarse-grained allocations appears to go a significant distance towards mitigating the negative performance impact of the strided memory accesses seen in the scalar ray pipelines.

This is consistent with both the findings of our investigation into HSAIL vector performance (section 5.4.4), where coarse-grained memory appeared relatively unaffected by strided stores while fine-grained memory performance was negatively impacted; and with the binary tree search benchmark (figure 5-8 and table 5-4) where fine-grained memory performed poorly with respect to a pattern of near-random memory reads, such as we might expect to see during a tree search.

A further consideration with respect to coarse-grained memory allocations relates to HSA's restrictions on concurrent access and to the transference of ownership between agents. In cases where a pair of tasks are scheduled to execute on different agents, it may be necessary to introduce additional nodes into the task graph to ensure that access permissions are updated correctly. This issue is only applicable during

transitions between agents, and not to dependent tasks resident on the same agent. Figure 6-13 provides an illustration of this.


**Figure 6-13:** Transferring Coarse-Grained Memory Access in RTKit

Similarly, if a task executing on PU 1 outputs results to a memory region that is inaccessible to a dependent task executing on PU 2, then a memory copy node can be inserted to move results to an accessible location.

### 6.3.10 Ray Compaction

In all of the preceding ray casting benchmarks, every stage of the pipeline was executed for each ray generated. Even in cases where a primary ray fails to intersect with any geometry, the shading and accumulation stages were still executed. This results in the execution of unnecessary shading and accumulation operations that do not contribute to the final image.

An alternative to this is to compact the sparse set of intersection points generated from the ray-surface intersection stage, into a densely packed set. This compaction from a sparse set of intersection points to a dense set has a non-zero computational cost. In order to provide a net positive benefit, the computational cost of performing this compaction must be smaller than the savings provided by eliminating unnecessary shading and accumulation. This is both scene and hardware dependent. In this benchmark, we aim to quantity this computational cost, and the potential benefits of applying compaction.

To accomplish this, we can introduce an additional ray compaction stage after intersection testing, but before shading. Figure 6-14 illustrates a task graph modified to introduce such a compaction stage, while listing 6-9 illustrates how this graph is expressed in RTKit.

For simplicity, we implement the compaction stage used in this example solely on the CPU. This further highlights the manner in which RTKit is able to make use of heterogeneous PUs. For the pipeline shown in figure 6-14, the ray identifier production and compaction stages were implemented solely on the CPU, while the remaining stages support both CPU and GPU implementations. This also serves to illustrate how RTKit enables us to rapidly explore optimization. Whilst compaction can reasonably

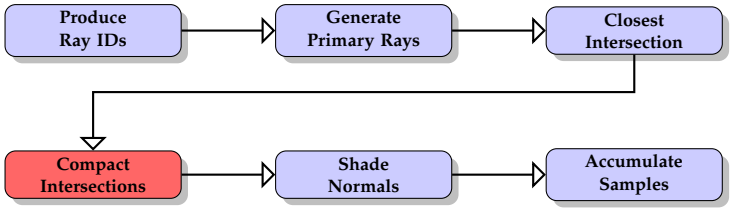



**Figure 6-14:** Task Graph for Ray Casting - Nearest Ray-Surface Intersection with Compaction

be implemented as a GPU kernel, a parallel implementation is more complex that required for a sequential CPU. In this example, we are able to quickly explore the impact of compaction through the introduction of a simple CPU implementation, before having to commit time and resources to providing a parallel implementation.

```
// Construct generate rays and identify ray-surface intersections as
// per previous examples.
auto ids  = make_task<ray_id_producer_t>();
auto rays = make_task<primary_ray_producer_t>(permuted_ids, ...);
auto sparse_hits = make_task<nearest_intersector_t>(rays, ...);

// Compact the sparse set of intersections into a dense set.
// For simplicity, we always execute this task on the CPU.
auto dense_hits = make_task<primitive_compaction_t>(sparse_hits,
                                                    ...);

// Continue the remainder of the pipeline as per the nearest hit
// benchmark.
auto shading = make_task<shade_normals_t>(dense_hits, ...);
auto accumulator = make_task<accumulator_t>(shading, ...);
```

**Listing 6-9:** Modified Ray Casting Pipeline for Compaction of Ray-Surface Intersections in RTKit

For brevity, we restrict this benchmark to the highest performing CPU and GPU pipelines from the previous examples: 8-wide CPU and 64-wide GPU packetized implementations. Figure 6-15 and table 6-11 illustrate the impact of introducing a ray compaction step prior to shading. As we might expect, the introduction of this step introduces additional costs in the Cornell Box, San Miguel and Sponza scenes. These are scenes where almost all primary rays can be expected to intersect with a surface. As a result, the set of ray-surface intersections is already relatively dense, and so compaction is unable to eliminate significant additional computation for later stages.

By contrast, the introduction of compaction provides performance improvements for the Lucy and Dragon scenes. This is primarily due to the large regions of the generated images for these scenes that contain no geometry. Compaction allows us to skip the shading and accumulation stages for these empty regions. As we noted during our analysis in section 6.3.4, the accumulation stage is effectively a sequential scatter operation on our CPU, and so we see large overall gains from limiting the quantity of data needing accumulation.

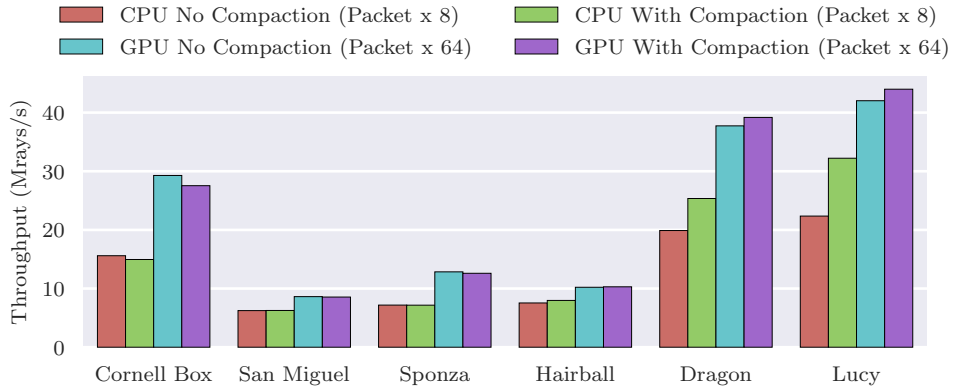



**Figure 6-15:** Coherent GPU Ray Casting Throughput - Compaction - Nearest Ray-Surface Intersection

| Scene | Cornell Box | San Miguel | Sponza | Hairball | Dragon | Lucy |
|---|---|---|---|---|---|---|
| **CPU (Packet x 8) - Nearest Ray-Surface Intersection** | | | | | | |
| No Compaction | 15.60 | 6.24 | 7.18 | 7.55 | 19.89 | 22.36 |
| With Compaction | 14.96 | 6.27 | 7.17 | 7.98 | 25.35 | 32.22 |
| **GPU (Packet x 64) - Nearest Ray-Surface Intersection** | | | | | | |
| No Compaction | 29.28 | 8.63 | 12.85 | 10.22 | 37.74 | 42.02 |
| With Compaction | 27.54 | 8.55 | 12.60 | 10.30 | 39.18 | 43.99 |

**Table 6-11:** Coherent GPU Ray Casting Throughput - Compaction (Mrays/s)

### 6.3.11 GPU Tree Traversal Comparisons

In this section, we describe the evaluation of a number of GPU BVH traversal algorithms implemented within RTKit. Previously, the majority of evaluations of GPU BVH traversal algorithms have been performed using discrete NVIDIA GPUs. To our

knowledge, none have been presented using AMD APUs. Given that discrete GPUs typically feature higher memory bandwidth and wider buses than the APU in our evaluation system, it was unclear how applicable previous evaluations were to our hardware.

In this benchmark, we measure the throughput of a range of BVH traversal algorithms, with the aim of establishing the most appropriate traversal algorithm for the integrated GPU found in our evaluation system.

Ray-surface intersections within RTKit are accelerated by performing a depth-first search on a BVH tree. Leaves within the tree contain scene geometry, while internal nodes consist of a hierarchy of nested AABBs. To find a ray-surface intersection, we begin at the tree root and iteratively test the ray against the AABBs of the children of the current node, with the search descending into nodes where the ray intersects the child AABB. When this search reaches a leaf, or the ray fails to intersect the children of an internal node, the search must backtrack.

For CPU ray tracing, a stack is commonly used to enable this backtracking. This technique has also been applied to GPU ray tracing by Aila and Laine (2009). However, this requires a large quantity of per-ray state when applied to massively parallel processors such as GPUs.

In order to limit the high state cost of maintaining per-ray stacks, a number of authors have explored stackless or short-stack tree search algorithms. Laine (2010) describes a technique based on maintaining a bit trail to track progress through the tree. As tree traversal progresses, a single 64-bit value is updated via bit manipulation. When this algorithm must backtrack, it instead restarts the search from the tree root, and utilizes the bit trail to enable the skipping of previously searched portions of the tree. This technique can be further coupled with a small stack. In this case, the stack can be used to backtrack short distances within the tree, and the restart bit trail used in cases where backtracking exhausts the stack. Laine further demonstrates how even a one or two element stack can significantly reduce the cost of backtracking.

Hapala et al. (2011) provide a technique which augments BVH nodes with pointers to their parent nodes in order to backtrack, coupled with a state-machine which calculates the next node be to visited based only on the ray direction, current node and the previously visited node. However, unlike the stack-based approach, this approach will revisit previously visited nodes while backtracking.

Barringer and Akenine-Möller (2013) describe three variants of a stackless BVH tree traversal algorithm. These are denoted *implicit-a*, *implicit-b*, and *sparse*. The two

implicit variants are based on representing binary trees as arrays. For any given node, the locations of the parent and child nodes can then be calculated based on the array index of the current node. This enables Barringer and Akenine-Möller to avoid the need to store pointers within tree nodes. However, these two variants leave empty space within the array in cases where the acceleration tree is not balanced, resulting in increased memory usage. The *sparse* third variant does not suffer from this limitation, but requires a each node to store a pointer to its parent.

In addition to the canonical stack-based algorithms, RTKit provides implementations each of these three techniques, including all three variants described by Barringer and Akenine-Möller. However, the BVH construction algorithm used within RTKit does not guarantee balanced trees, and currently lacks support for tree rebalancing. As a result of this limitation, the use of Barringer and Akenine-Möller's implicit variants is only practical on low-polygon scenes within RTKit. Due to the size of the example scenes used in our benchmarks, and our lack of rebalancing, we exclude results for the *implicit-a* and *implicit-b* variants. Further work on BVH tree balancing within RTKit is required before these two variants realistically can be utilized on non-trivial scenes.

Aila and Laine's traversal algorithms (Aila and Laine, 2009) located per-work-item stacks in private memory. We evaluate this approach, along with a variant which locates stacks in the higher bandwidth but limited capacity group memory. We additionally evaluate the three stackless techniques described above. For Laine's bit trail algorithm, we provide variants with a range of short-stack sizes.

Figure 6-16 and table 6-12 illustrate the performance of these various BVH traversal algorithms running on the integrated GPU in our evaluation system. It is important to highlight a subtle difference between the results presented here and previous benchmarks. Where previous benchmarks have described the throughput of a complete pipeline, these benchmarks solely isolate the performance of the ray-surface intersection task.

We will begin by examining the performance of the two stack-based approaches. Despite group memory offering higher bandwidth and lower latency than private memory in our evaluation system, the limited quantity of group memory and comparatively large state size of maintaining per-work-item stacks limits the amount available parallelism, resulting in poor performance. However, stack-based traversal performs well when stacks are located in private memory.

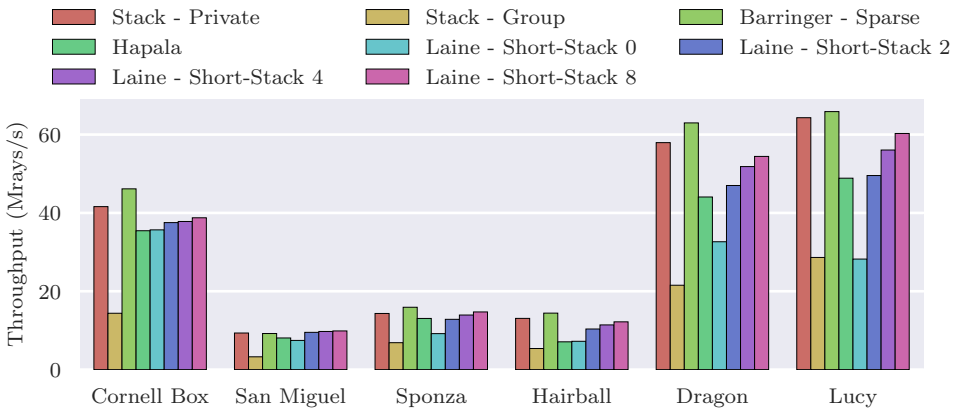



**Figure 6-16:** Coherent GPU Ray Casting Throughput - BVH Traversal - Nearest Ray-Surface Intersection

| Scene | Cornell Box | San Miguel | Sponza | Hairball | Dragon | Lucy |
|---|---|---|---|---|---|---|
| Stack - Private | 41.61 | 9.34 | 14.33 | 13.08 | 57.94 | 64.31 |
| Stack - Group | 14.39 | 3.27 | 6.87 | 5.39 | 21.54 | 28.63 |
| Barringer - Sparse | 46.14 | 9.22 | 15.92 | 14.42 | 62.98 | 65.86 |
| Hapala | 35.45 | 8.08 | 13.06 | 7.10 | 44.07 | 48.87 |
| Laine - Short-Stack 0 | 35.67 | 7.46 | 9.19 | 7.23 | 32.64 | 28.22 |
| Laine - Short-Stack 2 | 37.53 | 9.52 | 12.84 | 10.37 | 47.00 | 49.54 |
| Laine - Short-Stack 4 | 37.82 | 9.74 | 13.93 | 11.41 | 51.83 | 56.06 |
| Laine - Short-Stack 8 | 38.76 | 9.88 | 14.72 | 12.20 | 54.42 | 60.28 |

**Table 6-12:** Coherent GPU Ray Casting Throughput - BVH Traversal - Nearest Ray-Surface Intersection (Mrays/s)

When the Laine bit trail variants (Laine, 2010) are compared to each other, we find results that are broadly compatible with Laine's own results for NVIDIA GPUs, and with the CPU results presented by Barringer and Akenine-Möller (2013). Most notably, restarting tree traversal from the root in order to backtrack carries a high cost. This cost can be partially mitigated by the addition of a small stack. Here, even a small two element stack provides large benefits, with deeper stacks providing diminishing returns.

In comparing short-stack bit trail variants to the private memory full-stack variant, our results for most scenes remain consistent with those of both Barringer and Akenine-Möller, and Laine. In neither Barringer and Akenine-Möller's results for CPU, or Laine's results for NVIDIA GPUs did the performance of any of the bit trail algorithms exceed that of a deep stack, although both authors report comparatively small differences as the size of the short-stack is increased. However, for the two architectural scenes (San Miguel and Sponza), we do observe a small performance lead for the larger short-stack variants.

We find that Hapala et al.'s (2011) stackless algorithm performs relatively poorly across the board. Whilst it does outperform Laine's bit trail algorithm in the absence of a short-stack, even a two-element stack is generally sufficient to eliminate any advantage offered by the Hapala et al. algorithm. This can be attributed to the fact that Hapala et al.'s algorithm is forced to revisit internal nodes whilst backtracking, and incurs additional bandwidth and computational costs as a consequence.

Finally, we find that the *sparse* variant of Barringer and Akenine-Möller's stackless traversal algorithm delivers excellent performance for all scenes. It is able to outperform the stack-based approach on five out of six evaluation scenes. This is somewhat contrary to Barringer and Akenine-Möller's CPU results, which saw a 5-7% reduction rendering times when favouring a stack over their *sparse* stackless algorithm.

## 6.4 LIMITATIONS AND FUTURE WORK

As we saw in section 6.3.9, our scheduler lacks the ability to track limited resources such as coarse-grained memory. As a result, we exhausted the available coarse-grained memory on two evaluation scenes. Our task scheduler could be extended to track these resource limitations and throttle the spawning of additionally tasks until sufficient resources become available.

Also related to our scheduler is the problem of mapping tasks to specific PU. RTKit is currently designed to support the explicit mapping of tasks to particular PUs. This is in line with our original goal of enabling developers to explore the performance implications of specific mappings of tasks and PU. However, this developer guided mapping results in a static mapping that is unable to account for system load. In the most extreme mappings, such as the CPU-only and GPU-only mappings shown in section 6.3.6, this leaves other PUs under-utilized. Given that Offload and RTKit are able to generate multiple representations of tasks, dynamic load balance based on system load may provide an interesting avenue for further research.

Our experiments only demonstrate extremely simple shading, and consequently are likely to undervalue the importance of an efficient shading stage. This is not due to a fundamental limitation of our model, but rather due to the need to provide a more complete library of mathematical functions. Whilst our mathematical library provides a solid foundation for geometric calculations, it lacks sufficient support to implement state of the art shading algorithms. In particular, we lack adequate support for power and exponential functions. This ultimately relates back to an early observation that we made regarding the focus of HSA. OpenCL provides a rich library of built-in mathematical functions, while HSA expects that the developers of parallel languages will provide such implementations themselves, based on the particular requirements of their users. We previously encountered this issue whilst benchmarking Black-Scholes performance in section 5.7.7. In order to provide comparisons between CPU and GPU implementations, we require implementations of power and exponential functions for both SSE or AVX, and an implementation based on HSAIL's comparatively small set of native floating-point instructions.

Thus far, the construction of BVH trees has not been an area of focus for RTKit. Instead RTKit makes use of a simplistic multi-core BVH construction algorithm, and consequently restrict RTKit to static scenes. However, BVH tree construction is an area where the increased parallelism provided by an integrated GPU, and elimination of data movement and relatively low overhead of kernel dispatch may prove able to provide further improvements.

In introducing this chapter, we alluded to a goal of providing a toolkit that would allow for the evaluation of a variety of ray tracing algorithms and data structures across a range of heterogeneous PU. This was a sound goal when the work was conceived and conducted i.e. during the development of the HSA specifications. Many embedded and mobile SoC manufacturers were active contributors to the development of the HSA specifications. Unfortunately, no HSA runtime implementations

for such platforms have been made publicly available, and so this aspect remains largely unrealised. At the time of writing RTKit has been primarily evaluated on the CPU and GPU cores found in AMD Kaveri APUs, along with some preliminary investigations into performance on discrete AMD GPUs. Given that Offload, and by implication RTKit, targets HSA as defined by the specifications, initial evaluation on future HSA hardware should be a relatively straightforwards task.

## 6.5 DISCUSSION

Throughout this thesis, we aim to explore approaches to aiding domain experts in fields such as image processing or ray tracing to utilize the performance of heterogeneous systems. In chapter 4, we described a technique for building a domain-specific language for image processing upon SYCL. This approach was able to provide performance benefits over OpenCV, and deliver similar performance to Halide, on our evaluation system. However, due to SYCL's compilation model and the static nature of expression templates, retargeting this DSL for use on architecturally dissimilar systems would require the intervention of a machine expert.

In this chapter we have explored RTKit, our library for ray tracing on heterogeneous systems which builds upon the compiler and runtime described in chapter 5. This framework is partially motivated by a similar desire to provide tools to enable domain experts to exploit the performance of heterogeneous systems. However, in this case we aim to provide a framework to enable the exploration of the performance impact of various algorithmic choices and hardware mappings. Where our DSL-based solution adopted a relatively static approach to providing performance out-of-the-box, RTKit instead aims to provide toolkit for performance exploration. This difference in focus was motivated in part by our early involvement in the development of HSA itself. In this context, access to evaluation hardware, performance metrics and established optimization guidance was limited. This makes the use of static mappings based on machine expertise a less suitable approach for this use case.

In introducing this chapter, we defined three research questions:

- Is our programming model sufficient to implement a ray tracing pipeline?
- Can such ray tracing pipelines be implemented in a heterogeneous manner?

- Does doing so provide a performance benefit?

Throughout the course of this chapter, we have explored a series of examples of the use of RTKit, demonstrating both that the programming model described in chapter 5 provides sufficient foundations to enable the development of such a framework, and that we can utilize our framework to implement ray tracing algorithms which make use of heterogeneous PUs.

We were able to implement a variety of ray-tracing algorithms solely within our programming model. We have explored implementing portions of ray-tracing algorithms on both the CPU and integrated GPU, along with the performance of a variety of stack and stackless traversal algorithms on the GPU. Additionally, we have illustrated how the performance of workloads can vary based on the executing PU; the layout of data structures; the coherency properties of memory; and properties of the input data. Collectively, this amounts to considerable complexity for developers looking to map ray tracing to heterogeneous hardware. Our results suggest no simple answer to the challenge of finding a generally performant mapping of PU to the task of ray tracing, at least in the context of our evaluation system. Indeed, such a mapping is likely to be system dependent.

Given this complexity, optimizing for heterogeneous systems is a challenging task, with many competing factors. To help tackle this complexity, RTKit enables both the exploration of the mapping of tasks to PU through the use of a task graph, and the exploration of the impact of data layouts and vectorization through the extensive use of the C++ template system. In this way, RTKit enables the rapid exploration of the impact of each of these properties.

In motivating this thesis, we asserted both that heterogeneous systems have lead to a rise in complexity for software developers, and that high-level languages such as C++ could aid in tackling this complexity. This chapter has served as validation for those claims.

Finally, RTKit and our Offload compiler described in chapter 5 were developed concurrently. The benchmarks shown in chapter 5 were all relatively simple examples, primarily derived from existing OpenCL benchmarks. These allowed us to compare the performance of Offload and HSA relative to OpenCL, but did not demonstrate the use of Offload in a larger application. This chapter, and RTKit in general, serve as an illustrative example of such an application.

# Part IV

# Summary & Conclusions

# 7 CONCLUSIONS

The rise of both multi-core and heterogeneous systems has been coupled with rising complexity in programming such systems. Multi-core processors forced software developers to address concurrency and parallelism. Heterogeneous systems have retained these requirements. Beyond this, heterogeneous system impose further complexity. Architectural differences between Processing Units (PUs) in a heterogeneous system require the mapping of computational work to the most suitable PU; and complex segmented memory systems require the careful management of data movement and locality.

Today heterogeneous systems are pervasive. Accelerator devices and co-processors are common in the fields of High Performance Computing (HPC) and supercomputing; personal computers contain both general-purpose Central Processing Units (CPUs) and programmable Graphics Processing Units (GPUs); and modern smartphones typically contain an array of specialized CPU, GPU, Digital Signal Processor (DSP) and fixed-function PUs. This trend seems unlikely to disappear in the short term, Instead, we foresee continuing integration of additional specialized processing cores such as machine vision (Barry et al., 2015) and tensor (Jouppi et al., 2017) processors.

Over the course of this thesis, we have explored two approaches to applying heterogeneous systems to problems from the field of visual computing. This is comprised of three technical works:

- An approach to building a Domain Specific Language (DSL) for image processing on heterogeneous devices, based upon SYCL.
- Offload: A C++ compiler and programming model for Heterogeneous System Architecture (HSA), providing more advanced capabilities than SYCL.
- RTKit: a C++-based ray tracing framework for exploring performance optimization on heterogeneous systems, based on Offload.

These works share a number of common themes:

- Providing early usage experience and validation to two new standards for heterogeneous computing: SYCL and HSA
- The development and application of new C++ programming models for heterogeneous computing.
- The use of those programming models to build domain-specific toolkits for problems in the field of visual computing.

All three of the technical works presented in this thesis deal with the development of abstractions to ease the efficient use of heterogeneous systems for programmers with backgrounds outside the field of heterogeneous computing. In each case, these projects involved the use of proposed future standards for heterogeneous computing (SYCL and HSA) concurrent to the development of the respective standards, and prior to their public release.

Despite the pervasiveness of heterogeneous systems, programming models for heterogeneous systems are highly fragmented, and dominated by proprietary offerings such as CUDA.

C++ is potentially well-positioned to provide a common, standardized programming environment for such systems. C++ combines high-level language features such as templates, and low-level tools necessary for systems programming. However, there remain some fundamental questions that must be resolved in order to better align the standardized form of ISO C++ with the needs of heterogeneous systems. We address some of these issues in section 7.4.

## 7.1 THESIS SUMMARY

Over the course of this thesis, we have explored the implementation and use of single-source programming models based on C++, and applied these models to use cases from the fields of image processing and ray tracing.

In chapter 2, we provided a brief introduction to heterogeneous systems. We also introduced SYCL and HSA, two recent standards for heterogeneous computing. These two standards provide the foundations for our work presented in the subsequent chapters. Finally, we explored the fundamental algorithmic properties of our use cases: image processing and ray tracing; and how these properties interact with the

kernel execution model found in General Purpose Graphics Processing Unit (GPGPU) frameworks such as OpenCL.

We began chapter 3 by providing a survey of major frameworks for heterogeneous computing. We then reviewed relevant works on image processing on heterogeneous systems; C++ compilers for heterogeneous systems; compilers and runtimes targeting HSA specifically; and on ray tracing on heterogeneous systems.

In chapter 4, we demonstrated how the C++-based programming model provided by SYCL can be used to construct an embedded DSL for image processing. This DSL utilizes SYCL to transparently provide hardware acceleration on OpenCL 1.2 devices. For algorithmically similar kernels, we achieved equivalent performance to OpenCV and Halide for single operators. For larger pipelines, we are able to leverage the metaprogramming functionality of C++ to transparently compose single kernels from multiple operators. This results in improved performance compared to OpenCV, and required significantly less implementation effort than Halide. This work also represents one of the earliest evaluations of SYCL itself.

Both Halide and our DSL are pure functional languages, with a primary focus on representing functional mappings from one or more input images to output images. They have limited support for data structures beyond multi-channel images, limited control-flow, and functions may not have side-effects. Both approaches do allow for the execution of image-processing pipelines on heterogeneous processors via OpenCL. However, neither language is able to directly represent a pipeline that spans multiple heterogeneous processors, beyond via the construction of multiple independent pipelines, synchronized via host-code. This is distinct from our work on Offload in chapter 5, where we extend C++ to HSA. C++ is a more general-purpose language, allowing for the implementation of data structures and algorithms that would be challenging or impossible to implement in either Halide, or our DSL. Additionally, our programming model allows for fine-grained inter-device communication. However, this comes at the cost of requiring a much deeper understanding of heterogeneous systems, potentially making Offload less appropriate for image-processing users.

Our DSL builds upon SYCL. The model provided by SYCL enables the utilization of shared-source C++ on OpenCL-compatible accelerator devices. This represents a significant step forwards in terms of both bringing the power of template metaprogramming to OpenCL devices; and in easing the integration of host and device code on such platforms. SYCL abstracts the explicit management of data movement

required by OpenCL through a system of accessors and buffers. Constraints on the lifetimes of these accessors, coupled with the limitations on valid data types imposed by SYCL, require that data structures used on the host processor be transformed into an alternative form before use on accelerator devices. This is a significant limitation. However, it also enables SYCL to target devices with lower hardware requirements than Shared Virtual Memory (SVM)-based models such as HSA.

This limitation is rooted in the underlying model provided by OpenCL. OpenCL 1.2 provides dissimilar host and device representations of memory buffers, and no guarantee that the device-side address of a memory buffer will remain consistent across kernel dispatches. Furthermore, OpenCL 1.2 requires that any global memory buffers accessed from within a kernel are passed directly as kernel arguments, and not embedded within other data structures. Consequently, a single-source compiler targeting OpenCL must be able to identify the use of these buffers, and ensure that they are correctly passed as kernel arguments. In the general case, disambiguating pointers to global memory buffers from other pointers without additional annotations or metadata is a challenging task for a compiler. The use of special accessor types provides a compiler with the additional contextual information to simplify this task. Whilst OpenCL 2.0 supports SVM, most of the supported variants of SVM still require that a kernel dispatch either pass SVM pointers directly as kernel arguments, or that the programmer declare prior to dispatch the set of pointers that will be accessed from within the kernel, and so a similar construct to accessors is still required in these cases.

These limitations can be relaxed in the presence of a single unified address space, and a runtime environment that does not require this pre-declaration of SVM pointer accesses. In chapter 5, we explored a C++ compiler and runtime built upon HSA. This compiler and runtime are able to eliminate some of the constraints that we see in previous single-source programming models for C++ on heterogeneous systems, such as SYCL and C++ AMP. Most notably, the pervasive SVM found in HSA enables concurrent access to pointer-heavy data structures such as linked lists from multiple PUs.

Our programming model, combined with the fine-grained cache-coherent memory provided by HSA, is able to provide significant performance improvements for some workloads. The unified virtual address space and fine-grained coherence of system memory in HSA also enable a significant simplification over SYCL's model. The complex scheduler can be eliminated and the addresses of variables and data structures passed directly to kernel agents without the need for management of data move-

ment. This also serves to reduce both latency and memory bandwidth due to the elimination of copying between system agents.

The presence of cache-coherent memory in HSA also allows us to implement data structures that cannot currently be implemented in models such as SYCL or CUDA. The shared ring buffer illustrated in listing 5-1 provided an example of this.

Whilst our experiments utilizing fine-grained memory in HSA have demonstrated excellent performance for some workloads, they have also revealed cases where fine-grained memory performs extremely poorly. These cases typically involve either strided, or quasi-random memory access patterns.

Finally, we explored RTKit, a framework for ray tracing on heterogeneous systems in chapter 6. This new framework builds upon the compiler and runtime previously described in chapter 5. Whilst a significant body of work addresses ray tracing performance on accelerator devices such as GPUs, little work addresses shared memory heterogeneous systems, with recent work by Barringer, Andersson and Akenine-Möller (2016) being a notable exception.

RTKit serves the dual purpose of providing a convenient framework for exploring the optimization of ray-tracing algorithms on Accelerated Processing Units (APUs) and other heterogeneous systems, and in demonstrating the use of our C++ programming model for HSA in a non-trivial application. We were able to compare the relative performance of mapping tasks in a ray tracing pipeline to CPU and GPU cores, the impact of strided and packed data layouts, and of using memory allocations with coarse and fine-grained consistency.

## 7.2 LIMITATIONS

The restricted availability of both runtime implementations and supporting hardware platforms for both SYCL and HSA has proved a significant limiting factor in our work. This limited availability severely restricts our ability to generalize our conclusions beyond the hardware and runtime implementations currently available for evaluation.

Our work on both implementing DSLs on SYCL, and on our C++ programming model targeting HSA, began relatively early in the development of the respective

specifications. This was prior to the public release of implementations of either SYCL or HSA.

In the case of SYCL, two implementations were under active development during the course of this thesis: ComputeCpp (Codeplay Software Ltd., 2016) and triSYCL (Keryell, 2015). ComputeCpp (Codeplay Software Ltd., 2016) currently targets OpenCL 1.2 implementations with support for the Standard Portable Intermediate Representation (SPIR) extension. Consumption of SPIR is an optional feature for which many OpenCL implementations currently lack support. In practice, this restricts ComputeCpp to a number of platforms produced by Advanced Micro Devices (AMD) and Intel. This limitation can be resolved in the future through the addition of further device compiler backends targeting non-optional OpenCL input formats such as OpenCL C or SPIR-V to ComputeCpp. At the time of writing, the open-source triSYCL (Keryell, 2015) implementation is primarily CPU-only, although integration with precompiled Field-Programmable Gate Array (FPGA) kernels has also been demonstrated (Doumoulakis, Keryell and O'Brien, 2017). Due to CPU-only nature of triSYCL, and to differing levels of implementation progress between the ComputeCpp and triSYCL implementations, our work has yet to be evaluated on triSYCL. As the project matures, evaluation on triSYCL would provide an alternative data point and an interesting avenue for further investigation.

With respect to HSA, hardware availability is much more significantly constrained. ARM, Imagination Technologies and MediaTek have all publicly discussed plans for future HSA-compliant hardware (Nicholas, 2015). To date, AMD is the only hardware vendor to publicly release a runtime implementation. For much of the time period over which the work described in this thesis was conducted, hardware support was restricted to APUs. More recently AMD have released support for discrete GPUs (Advanced Micro Devices, 2016c). To date, we have conducted preliminary experiments to demonstrate support for discrete GPUs, but a more detailed evaluation remains future work. HSA was envisioned as a foundation upon which parallel programming models could build in order to access a plethora of hardware devices. At the time of writing, whilst we have demonstrated the suitability of HSA for the implementation of parallel programming models, the HSA Foundation's vision of support for a wide variety of hardware devices has not been realised. Consequently, it is difficult to make more general statements about the suitability of our programming model for more diverse hardware.

## 7.3 IMPLICATIONS AND IMPACT

Heterogeneous systems have seen a significant rise in adoption in recent years, both at scale in fields such as HPC and supercomputing, and in small, energy-efficient devices such as smartphones. This growth leads to a need for standardised and pervasive software interfaces and tools. As a result, there is a desire to bring support for heterogeneous accelerator devices to ISO C++ by 2020 [personal communication, M. Wong, 2017].

Whilst there remain many open questions with respect to C++ programming models for heterogeneous and many-core systems, several groups are actively working on promising approaches. There is substantial commonality between SYCL (Khronos OpenCL Working Group – SYCL subgroup, 2015), HCC/C++ AMP (Microsoft Corporation, 2013; Sander et al., 2015), PACXX (Haidl and Gorlatch, 2014), HPX (Kaiser, Heller et al., 2014), Kokkos (Edwards, Sunderland et al., 2012), RAJA (Hornung, Keasler et al., 2014), and our own work. The majority of these works focus on models for discrete GPUs, or on HPC and supercomputing scale problems. Our work provides an alternative data point, focused on smaller devices such as System-on-Chip (SoC) and APUs. With the notable exception of HCC, the aforementioned works have not addressed how features like the low-latency dispatch and shared virtual memory found in HSA might impact a heterogeneous C++ programming model. These properties are core to our work, and utilized heavily by RTKit.

Our work has produced impact in a number of ways:

1. Contributions to Codeplay Software IP
2. Contributions to Industry Standards
3. Peer-reviewed Publications
4. Dissemination to Academic and Industrial Audiences

### 7.3.1 Contributions to Codeplay Software IP

Ideas described within this thesis have subsequently been utilized and further explored by researchers within Codeplay Software. This is particularly true of the work on DSLs and kernel fusion described in chapter 4. Ideas derived from this work can be found in VisionCpp (Goli, 2016), a framework for accelerating computer vis-

ion operations on embedded hardware devices; SYCL-BLAS (Aliaga, Reyes and Goli, 2017a,b), a SYCL-based linear algebra library; and within the SYCL backend for Eigen, which forms a component of ongoing work to provide SYCL support within the TensorFlow library for machine learning (Goli, Iwanski and Richards, 2017).

These projects all share similar properties. They are C++ header file-only implementations, which are relatively easy to integrate into external projects. This makes introducing a dependency on an external tool such a Halide less desirable. Additionally in these domains, element-wise maps and reductions are common patterns, both of which are relatively straight-forwards to map onto both OpenCL's execution model and expression template based DSLs. However, it should be noted that whilst the use of expression templates to construct SYCL-based DSLs does provide a reasonable route to attaining the benefits of kernel fusion without additional external tools, it is significantly less flexible than an approach like Halide's with respect to hardware portability. In cases where there is a high probability of needing to optimize for a different hardware platform in the future, or for application domains where such map and reduction patterns are less prevalent, an alternative approach is likely to provide a stronger solution.

Our work on C++ programming models for HSA has both acted as a validation of the core technology within the Codeplay Offload compiler (P. Cooper et al., 2010; Donaldson et al., 2010), and resulted in extensions that increase the flexibility of the core compiler framework. At present, the primary use case for Offload is in supporting ComputeCpp (Codeplay Software Ltd., 2016), Codeplay Software's SYCL implementation.

### 7.3.2 Contributions to Industry Standards

Codeplay Software are deeply involved in a number of industry standards organizations, including the C++ Standards Committee, the Khronos Group and the HSA Foundation. The experience and knowledge gained throughout the development of this thesis has enabled contributions to both the SYCL and HSA standards, along with an extension to Vulkan (Khronos Vulkan Working Group, 2016).

In addition to ongoing working group participation throughout the development of the aforementioned standards, and contributing a range of smaller changes and clarifications, our work has resulted in more significant changes in a number of areas. These include relaxation of type constraints in SYCL to enable better support for

metaprogramming techniques such as expression templates, improvements to support for fine and coarse grained memory regions in HSA, and contributions to the addition of subgroup support to Vulkan 1.1.

Standards documents and specifications from organisations such as the Khronos Group and the HSA Foundation typically only become available to non-members when they are substantially complete. Furthermore, the release of implementations of these standards lags behind the development of the standards themselves. Consequently, practical experience of use is often limited during standards development.

Our work described in chapters 4 to 6 began whilst the initial designs of both SYCL and the HSA runtime Application Programming Interface (API) were still under development, and prior to the publication of drafts of either specification. We were able to provide feedback on both specifications during their development, resulting in a number of changes to both current and upcoming revisions of the specifications.

### 7.3.3 Dissemination

Our work has been disseminated to a variety of relevant audiences. These include both academic and industrial audiences, along with the open-source community of the Clang and LLVM compiler project. A full list of publications and presentations can be found in appendix A.

In addition to providing feedback on the specifications themselves, our work with both SYCL and HSA has been presented to the standards working groups for SYCL and HSA (Potter, 2015, 2016a,c), along with to SG14 (Potter, 2016b), the ISO C++ study group focused on addressing the needs of communities requiring on high-performance and low-latency C++. This serves to aid each of these groups in the design of future specifications.

Our HSA compiler has also been used in promotional materials and early performance evaluations by AMD prior to the development of Heterogeneous Compute Compiler (HCC), and results from this thesis have similarly been utilized by representatives of the HSA Foundation in promotional materials (Blinzer, 2016; Glossner, 2016).

## 7.4 TOWARDS THE UNIFICATION OF HETEROGENEOUS SYSTEMS AND ISO C++

The future model for C++ on heterogeneous accelerators remains undecided. Heterogeneous systems have become increasingly important in recent years, particularly in fields where performance is important. Many of these are also fields in which C++ has seen a high adoption rate such as computer games, finance and scientific computing. It seems likely that future versions of C++ will provide support for heterogeneous processors beyond the limited possibilities of parallel Standard Template Library (STL) (ISO/IEC, 2015).

The C++ specifications assume an underlying abstract machine (ISO/IEC, 2014, p. 8). Initially this was a single-processor machine executing a single thread of execution within a process. C++11 later updated this conceptual model to support multiple threads of execution, and consequently the memory model to define the visibility and ordering of memory operations. These changes advanced C++ sufficiently to address the needs of software developers targeting multi-core systems.

However, these changes are not sufficient to address the challenges of heterogeneous systems, and thus further updates are required. There remains significant discrepancies and incompatibilities between the current abstract machine model upon which the ISO C++ standard is based, and the kernel execution and memory models adopted by frameworks for heterogeneous computing. Our work represents a modest step towards defining a more unified model.

We believe that support for heterogeneous systems can also be introduced, but not without further extension and definition of the C++ abstract machine. In order to unify programming models for heterogeneous systems and ISO C++, we would need to address a number of issues:

1. Discovery and introspection of accelerator devices
2. Work dispatch to accelerators
3. Extend the memory model to support dedicated device memories
4. Addressing features of the C++ programming model not currently supported by hardware

Some of these discrepancies and incompatibilities are relatively tractable.

Functionality to enable the discovery and introspection of accelerator devices is likely to only require extensions to the C++ standard library, and not the language itself. This simply requires the definition of types representing PUs and their properties. Whilst earlier iterations on C++ entirely abstracted the underlying machine, C++11 provides some very limited precedent for this. In order to enable software developers to query the number of concurrent threads available on their platform, the function `std::thread::hardware_concurrency()` was introduced. This is a departure from the previous approach of abstracting away details on the underlying hardware, and this departure would have to be further expanded to enable developers to make informed decisions about where to execute work within a heterogeneous system.

The ability to query what PUs are present within a machine is only valuable when coupled with a mechanism to dispatch work to PUs. There appears to be some convergence on a model for this. Our own work; C++ AMP and HCC; and High Performance ParalleX (HPX) all support multiple heterogeneous PUs and all independently favour the use of futures to chain computations. The C++ executors proposal (Hoberock et al., 2017) may provide a route towards abstracting the kernel launch process. This proposal aims to provide mechanisms to control how and where asynchronous work is executed, and whilst it does not currently encompass support for dispatch to heterogeneous accelerators, it presents a promising starting point.

The remaining discrepancies are more challenging, requiring changes to either the C++ memory model or the core language itself.

The first such issue is the question of representing multiple physical memories, with differing performance characteristics. In some cases, these memories may not be accessible to all PUs directly. This is departure from the current C++ memory model.

The use of pervasive fine-grained cache-coherent SVM, such as that found in HSA potentially provides the most straight-forward route here. However, our benchmarks in both chapter 5 and chapter 6 demonstrated severe negative performance implications to the use of fine-grained SVM for certain workloads on our APU-based evaluation system. It seems likely that these negative performance issues would only be further exacerbated on discrete GPUs. As such, it seems likely that some model similar to HSA's, allowing for regions of memory with restricted accessibility or relaxed cache-coherency, may be necessary in order to retain performance.

Beyond the question of how regions of memory should be represented in the abstract machine, there are also fundamental differences between how models such as CUDA's unified memory or our own C++ programming model for HSA expose such memory to software developers, when compared to models that favour high-level abstractions such as C++AMP or SYCL. Our model utilizes pointers as the most basic primitive for addressing and manipulating memory, while SYCL utilizes a system of buffers and accessors.

The approach adopted by SYCL has the advantage of allowing a SYCL runtime implementation significant freedom to manage data movement internally, potentially even relocating data from host memory to dedicated memory located on an accelerator device. This freedom for a runtime to manage the locality of data imposes fewer requirements on the underlying hardware than the SVM-based approach favoured by our programming model. However, it comes at a cost for software developers, due to the challenges of integrating the necessary abstractions into a codebase.

A further challenging question is that of how, and whether, to integrate PU that are incapable of supporting the full capabilities of the C++ programming model. OpenCL 1.2 did not require accelerators to support function pointers, exceptions or runtime recursion, and this class of hardware is widespread.

Historically, heterogeneous programming models such as OpenCL or C++ AMP have resolved this issue by simply prohibiting the use of these features within kernels. However, this is an unsatisfying solution, especially if it leads to a situation where these features are valid in source code regions executed on a host processor, but prohibited on accelerator devices. More modern GPU hardware is able to address some of these challenges, with HSA providing support for function pointers. However, exceptions remain particularly challenging. Indeed, we can already see this issue arise within parallel STL, where a parallel algorithm is allowed to simply terminate an application on an exception, rather than following the standard exception handling mechanisms (ISO/IEC, 2015).

## 7.5 FUTURE RESEARCH DIRECTIONS

There remains ample scope for further investigation of all three of the topics addressed in the preceding chapters. In this chapter we will explore some further questions raised by the work described in chapters 4 to 6.

All three of the works described in this thesis have been concerned with the performance behaviour of heterogeneous systems.

There are plenty of opportunities for further research on language and library-based approaches to C++ on heterogeneous systems. The problems described in section 7.4 require generalizable and future-proof solutions suitable for standardization into ISO C++. In particular, the questions of how both complex memory hierarchies should be represented, and how PUs that lack the functionality to support the full C++ programming model can be integrated. There is also scope for exploring the development of C++ programming models for alternative platforms with additional constraints such as Vulkan (Khronos Vulkan Working Group, 2016).

In introducing our work on exploring the implementation of DSLs on SYCL, we asserted that such DSLs are capable of delivering a separation of concerns, such that domain experts can work with familiar syntax and conceptual models, while machine experts focus on delivering hardware-specific performance optimizations.

This raises the question of performance portability. We have provided a performance evaluation for our DSL on a single hardware platform, and whilst our implementation can be expected to execute correctly on any platform supporting SYCL, this provides no guarantee of achieving maximal performance. There is ample scope for both evaluation of our implementation on other hardware platforms providing OpenCL support, and for exploring the implications of combining our DSL with our HSA-based programming model.

Further related to the question of achieving predictable and portable performance is the question of identifying under what circumstances kernel fusion or fission should be applied. Whilst exploring the impact of generating fused kernels from primitives, we encountered situations where such fusion is undesirable. This might be due to such fusion introducing additional computation and bandwidth requirements, as in the case of separated versus non-separated convolutions, or due to generating increased pressure on finite resources such as registers or group memory.

The use of cost models to aid in identifying when fusion or fission of kernels should be performed appears to be a potentially valuable extension, particularly if such models could be made sufficiently expressive to highlight areas where the optimal optimization strategy might differ between hardware platforms. This is a complex task, requiring understanding of the hardware characteristics of specific PUs, the kernel execution model, and data flow within the algorithm upon which fusion is to be attempted. Whilst this problem seems particularly challenging in the general

case, problem domains such as image processing, linear algebra and convolutional neural networks have a constrained forms which may serve to make this problem more tractable.

A further question is whether the SYCL programming model possesses sufficient information to enable the application of kernel fusion at runtime, either automatically, or with the addition of modest additional metadata by application developers. The SYCL programming model possesses significant information about the dependencies between kernels and the runtime has considerable latitude in the scheduling of the execution of those kernels. This exposes the possibility of combining multiple kernels together at runtime. However, such an approach would require both a deeper analysis of the likely performance impact of fusing kernels and analysis of the kernel dependency graph held within a SYCL runtime.

# BIBLIOGRAPHY


Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I.J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D.G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P.A., Vanhoucke, V., Vasudevan, V., Viégas, F.B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X., 2016. Tensorflow: large-scale machine learning on heterogeneous distributed systems. *CoRR* [Online], abs/1603.04467. Available from: `http://arxiv.org/abs/1603.04467`.

*Adding HSA support to aparapi lambda branch*. `https://code.google.com/p/aparapi/wiki/HSAEnablementOfLambdaBranch`. Accessed: 29 November 2015.

Advanced Micro Devices, 2008. *AMD stream computing user guide*.

Advanced Micro Devices, 2016a. *Graphics Core Next Architecture, Generation 3*.

Advanced Micro Devices, 2016b. *RadeonRays*. AMD. Available from: `https://github.com/GPUOpen-LibrariesAndSDKs/RadeonRays_SDK`.

Advanced Micro Devices, 2016c. *ROCm - open source platform for HPC and ultrascale GPU computing* [Online]. [Accessed: 7 June 2017]. Available from: `https://github.com/RadeonOpenCompute/ROCm`.

Advanced Micro Devices. *APP SDK – a complete development platform - AMD*. Accessed: 3 June 2017. Available from: `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk`.

Aila, T. and Karras, T., 2010. Architecture considerations for tracing incoherent rays. In: J. Hensley, P. Slusallek, D.K. McAllister and C.P. Gribble, eds. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on high performance graphics 2010* [Online], 25–27 June 2010 Saarbrücken, Germany. Eurographics Association, pp.113–122. Available from: `http://dx.doi.org/10.2312/EGGH/HPG10/113-122`.

Aila, T. and Laine, S., 2009. Understanding the efficiency of ray traversal on gpus. In: S.N. Spencer, D.K. McAllister, M. Pharr, I. Wald, D.P. Luebke and P. Slusallek, eds. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on high performance graphics 2009* [Online], 1–3 August 2009 New Orleans, Louisiana, USA. Eurographics Association, pp.145–149. Available from: `http://dx.doi.org/10.2312/EGGH/HPG09/145-150`.

Aila, T., Laine, S. and Karras, T., 2012. *Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum*. NVIDIA Corporation, (NVIDIA Technical Report NVR-2012-02).

Akenine-Möller, T. and Ström, J., 2008. Graphics processing units for handhelds. *Proceedings of the IEEE* [Online], 96(5), pp.779–789. Available from: `http://dx.doi.org/10.1109/JPROC.2008.917719`.

Akhloufi, M. and Campagna, A., 2014. OpenCLIPP: OpenCL integrated performance primitives library for computer vision applications. *Proc. spie electronic imaging*, pp.25–31.

Aliaga, J.I., Reyes, R. and Goli, M., 2017a. SYCL-BLAS: combining expression trees and kernel fusion on heterogeneous systems. In: S. Bassini, M. Danelutto, P. Dazzi, G.R. Joubert and F.J. Peters, eds. *Parallel computing is everywhere, proceedings of the international conference on parallel computing, ParCo 2017* [Online], Bologna, Italy. Vol. 32, Advances in Parallel Computing. IOS Press, pp.349–358. Available from: `http://dx.doi.org/10.3233/978-1-61499-843-3-349`.

Aliaga, J.I., Reyes, R. and Goli, M., 2017b. SYCL-BLAS: leveraging expression trees for linear algebra. In: S. McIntosh-Smith and B. Bergen, eds. *Proceedings of the 5th international workshop on OpenCL, IWOCL 2017* [Online], 16–18 May 2017 Toronto, Canada. ACM, 32:1–32:5. Available from: `http://dx.doi.org/10.1145/3078155.3078189`.

Appel, A., 1968. Some techniques for shading machine renderings of solids. *American federation of information processing societies: AFIPS conference proceedings: 1968 spring joint computer conference* [Online], 30 April–2 May 1968 Atlantic City, NJ, USA. Vol. 32, AFIPS Conference Proceedings. Thomson Book Company, Washington D.C., pp.37–45. Available from: `http://dx.doi.org/10.1145/1468075.1468082`.

Apple, 2014. *Metal* [Online]. Available from: `developer.apple.com/documentation/metal`.

Barringer, R., Andersson, M. and Akenine-Möller, T., 2016. Ray accelerator: efficient and flexible ray tracing on a heterogeneous architecture. *Computer Graphics Forum* [Online], n/a–n/a. Available from: http://dx.doi.org/10.1111/cgf.13071.

Barringer, R. and Akenine-Möller, T., 2013. Dynamic stackless binary tree traversal. *Journal of Computer Graphics Techniques (JCGT)* [Online], 2(1) (), pp.38–49. Available from: http://jcgt.org/published/0002/01/03/.

Barry, B., Brick, C., Connor, F., Donohoe, D., Moloney, D., Richmond, R., O'Riordan, M.J. and Toma, V., 2015. Always-on vision processing unit for mobile applications. *IEEE Micro* [Online], 35(2), pp.56–66. Available from: http://dx.doi.org/10.1109/MM.2015.10.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I.J., Bergeron, A., Bouchard, N., Warde-Farley, D. and Bengio, Y., 2012. Theano: new features and speed improvements. *CoRR* [Online], abs/1211.5590. arXiv: 1211.5590. Available from: http://arxiv.org/abs/1211.5590.

Bastoul, C., 2004. Code generation in the polyhedral model is easier than you think. *13th international conference on parallel architectures and compilation techniques, PACT 2004* [Online], 29 September–3 October 2004 Antibes Juan-les-Pins, France. IEEE Computer Society, pp.7–16. Available from: http://dx.doi.org/10.1109/PACT.2004.10018.

Batcher, K.E., 1968. Sorting networks and their applications. *American federation of information processing societies: AFIPS conference proceedings: 1968 spring joint computer conference, atlantic city, nj, usa, 30 april - 2 may 1968* [Online]. Vol. 32, AFIPS Conference Proceedings. Thomson Book Company, Washington D.C., pp.307–314. Available from: http://dx.doi.org/10.1145/1468075.1468121.

Bawidamann, U. and Nehmeier, M., 2011. Expression templates and OpenCL. In: R. Wyrzykowski, J. Dongarra, K. Karczewski and J. Wasniewski, eds. *Parallel processing and applied mathematics - 9th international conference, PPAM 2011, revised selected papers, part II* [Online], 11–14 September 2011 Torun, Poland. Vol. 7204, Lecture Notes in Computer Science. Springer, pp.71–80. Available from: http://dx.doi.org/10.1007/978-3-642-31500-8_8.

Beck, S., Bernstein, A., Danch, D. and Fröhlich, B., 2005. CPU-GPU hybrid real time ray tracing framework.

Bell, N. and Hoberock, J., 2011. Thrust: A productivity-oriented library for CUDA. *GPU computing gems Jade edition*, 2, pp.359–371.

Benthin, C., Wald, I., Woop, S., Ernst, M. and Mark, W.R., 2012. Combining single and packet-ray tracing for arbitrary ray distributions on the Intel MIC architecture. *IEEE Trans. Vis. Comput. Graph.* [Online], 18(9), pp.1438–1448. Available from: http://dx.doi.org/10.1109/TVCG.2011.277.

Bentley, J.L., 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* [Online], 18(9), pp.509–517. Available from: http://dx.doi.org/10.1145/361002.361007.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D. and Bengio, Y., 2010. Theano: A CPU and GPU math compiler in Python. *Proc. 9th python in science conf*. Vol. 1.

Bernstein, G.L., Shah, C., Lemire, C., DeVito, Z., Fisher, M., Levis, P. and Hanrahan, P., 2016. Ebb: A DSL for physical simulation on cpus and gpus. *ACM Trans. Graph.* [Online], 35(2), p.21. Available from: http://dx.doi.org/10.1145/2892632.

Bikker, J. and Schijndel, J. van, 2013. The brigade renderer: A path tracer for real-time games. *Int. J. Computer Games Technology* [Online], 2013, 578269:1–578269:14. Available from: http://dx.doi.org/10.1155/2013/578269.

Blinzer, P., 2016. *The heterogeneous system architecture it's beyond the GPU*.

Brand, S., 2017. *Hsa-debugger-lldb-source*. Accessed: 16 June 2017. Available from: https://github.com/HSAFoundation/HSA-Debugger-LLDB-Source.

Branover, A., Foley, D. and Steinman, M., 2012. AMD fusion APU: Llano. *IEEE Micro* [Online], 32(2), pp.28–37. Available from: http://dx.doi.org/10.1109/MM.2012.2.

Briggs, P., Cooper, K.D. and Torczon, L., 1992. Rematerialization. In: S.I. Feldman and R.L. Wexelblat, eds. *Proceedings of the ACM SIGPLAN'92 conference on programming language design and implementation, PLDI 1992* [Online], 17–19 June 1992 San Francisco, California, USA. ACM, pp.311–321. Available from: http://dx.doi.org/10.1145/143095.143143.

Brown, K.J., Sujeeth, A.K., Lee, H., Rompf, T., Chafi, H., Odersky, M. and Olukotun, K., 2011. A heterogeneous parallel framework for domain-specific languages. In: L. Rauchwerger and V. Sarkar, eds. *2011 international conference on parallel architectures and compilation techniques, PACT 2011* [Online], 10–14 October 2011 Galveston, Texas, USA. IEEE Computer Society, pp.89–100. Available from: `http://dx.doi.org/10.1109/PACT.2011.15`.

Buck, I., Foley, T., Horn, D.R., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P., 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.* [Online], 23(3), pp.777–786. Available from: `http://dx.doi.org/10.1145/1015706.1015800`.

Campbell, C. and Miller, A., 2011. *Parallel programming with microsoft visual c++: design patterns for decomposition and coordination on multicore architectures*. 1st. Microsoft Press.

Carr, N.A., Hall, J.D. and Hart, J.C., 2002. The ray engine. In: T. Ertl, W. Heidrich and M.C. Doggett, eds. *Proceedings of the 2002 ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware* [Online], 2–3 September 2002 Saarbrücken, Germany. The Eurographics Association, pp.37–46. Available from: `http://dx.doi.org/10.2312/EGGH/EGGH02/037-046`.

Cerveny, V., 1985. The application of ray tracing to the numerical modeling of seismic wavefields in complex structures. *Seismic shear waves*, 15, pp.1–124.

Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M. and Olukotun, K., 2010. Language virtualization for heterogeneous parallel computing. In: W.R. Cook, S. Clarke and M.C. Rinard, eds. *Proceedings of the 25th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA 2010* [Online], 17–21 October 2010 Reno/Tahoe, Nevada, USA. ACM, pp.835–847. Available from: `http://dx.doi.org/10.1145/1869459.1869527`.

Chakrabarti, G., Grover, V., Aarts, B., Kong, X., Kudlur, M., Lin, Y., Marathe, J., Murphy, M. and Wang, J., 2012. CUDA: compiling and optimizing for a GPU platform. In: H.H. Ali, Y. Shi, D. Khazanchi, M. Lees, G.D. van Albada, J. Dongarra and P.M.A. Sloot, eds. *Proceedings of the international conference on computational science, ICCS 2012* [Online], 4–6 June 2012 Omaha, Nebraska, USA. Vol. 9, Procedia Computer Science. Elsevier, pp.1910–1919. Available from: `http://dx.doi.org/10.1016/j.procs.2012.04.209`.

Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L. and Grover, V., 2011. Accelerating haskell array codes with multicore gpus. In: M. Carro and J.H. Reppy, eds. *Proceedings of the POPL 2011 workshop on declarative aspects of multicore programming, DAMP 2011* [Online], 23 January 2011 Austin, Texas, USA. ACM, pp.3–14. Available from: http://dx.doi.org/10.1145/1926354.1926358.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S. and Skadron, K., 2009. Rodinia: A benchmark suite for heterogeneous computing. *Proceedings of the 2009 IEEE international symposium on workload characterization, IISWC 2009* [Online], 4–6 October 2009 Austin, Texas, USA. IEEE Computer Society, pp.44–54. Available from: http://dx.doi.org/10.1109/IISWC.2009.5306797.

Che, S., Sheaffer, J.W. and Skadron, K., 2011. Dymaxion: optimizing memory access patterns for heterogeneous systems. In: S. Lathrop, J. Costa and W. Kramer, eds. *Conference on high performance computing networking, storage and analysis, SC 2011* [Online], 12–18 November 2011 Seattle, WA, USA. ACM, 13:1–13:11. Available from: http://dx.doi.org/10.1145/2063384.2063401.

Chen, C. and Liu, D.S., 2007. Use of hardware z-buffered rasterization to accelerate ray tracing. In: Y. Cho, R.L. Wainwright, H. Haddad, S.Y. Shin and Y.W. Koo, eds. *Proceedings of the 2007 ACM symposium on applied computing, sac 2007* [Online], 11–15 March 2007 Seoul, Korea. ACM, pp.1046–1050. Available from: http://dx.doi.org/10.1145/1244002.1244231.

Chen, L. and Agrawal, G., 2012. Optimizing mapreduce for GPUs with effective shared memory usage. In: D.H.J. Epema, T. Kielmann and M. Ripeanu, eds. *The 21st international symposium on high-performance parallel and distributed computing, HPDC 2012* [Online], Delft, Netherlands. ACM, pp.199–210. Available from: http://dx.doi.org/10.1145/2287076.2287109.

Chiw, C., Kindlmann, G.L., Reppy, J.H., Samuels, L. and Seltzer, N., 2012. Diderot: a parallel DSL for image analysis and visualization. In: J. Vitek, H. Lin and F. Tip, eds. *ACM SIGPLAN conference on programming language design and implementation, PLDI 2012* [Online], 11–16 June 2012 Beijing, China. ACM, pp.111–120. Available from: http://dx.doi.org/10.1145/2254064.2254079.

Choi, H., Choi, W., Quan, T.M., Hildebrand, D.G.C., Pfister, H. and Jeong, W., 2014. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE Trans. Vis. Comput. Graph.* [Online], 20(12), pp.2407–2416. Available from: http://dx.doi.org/10.1109/TVCG.2014.2346322.

Codeplay Software Ltd., 2016. *ComputeCpp*. Available from: https://codeplay.com/products/computecpp.

CodeSourcery, Compaq, EDG, HP, Red Hat and SGI, 2004. *Itanium C++ ABI* [Online]. Accessed: 29 November 2015. Available from: http://www.codesourcery.com/public/cxx-abi/.

Cole, M., 1988. *Algorithmic skeletons: A structured approach to the management of parallel computation*. PhD thesis. University of Edinburgh.

Collobert, R., Kavukcuoglu, K. and Farabet, C., 2011. Torch7: a matlab-like environment for machine learning. *Biglearn, nips workshop*, EPFL-CONF-192376.

Cook, R.L., Porter, T.K. and Carpenter, L.C., 1984. Distributed ray tracing. In: H. Christiansen, ed. *Proceedings of the 11th annual conference on computer graphics and interactive techniques, SIGGRAPH 1984* [Online], Minneapolis, MA, USA. ACM, pp.137–145. Available from: http://dx.doi.org/10.1145/800031.808590.

Cooper, P., Dolinsky, U., Donaldson, A.F., Richards, A., Riley, C. and Russell, G., 2010. Offload - automating code migration to heterogeneous multicore systems. In: Y.N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi and X. Martorell, eds. *High performance embedded architectures and compilers, 5th international conference, HiPEAC 2010, proceedings* [Online], 25–27 January 2010 Pisa, Italy. Vol. 5952, Lecture Notes in Computer Science. Springer, pp.337–352. Available from: http://dx.doi.org/10.1007/978-3-642-11515-8_25.

Cornwall, J.L.T., Beckmann, O. and Kelly, P.H.J., 2006. Automatically translating a general purpose C++ image processing library for GPUs. *20th international parallel and distributed processing symposium, IPDPS 2006, proceedings* [Online], 25–29 April 2006 Rhodes Island, Greece. IEEE. Available from: http://dx.doi.org/10.1109/IPDPS.2006.1639716.

Cornwall, J.L.T., Howes, L.W., Kelly, P.H.J., Parsonage, P. and Nicoletti, B., 2009. High-performance SIMT code generation in an active visual effects library. In: G. Johnson, C. Trinitis, G. Gaydadjiev and A.V. Veidenbaum, eds. *Proceedings of the 6th conference on computing frontiers, 2009* [Online], 18–20 May 2009 Ischia, Italy. ACM, pp.175–184. Available from: http://dx.doi.org/10.1145/1531743.1531772.

Cornwall, J.L.T., Kelly, P.H.J., Parsonage, P. and Nicoletti, B., 2007. Explicit dependence metadata in an active visual effects library. *Languages and compilers for parallel computing, 20th international workshop, LCPC 2007, revised selected papers* [Online], 11–13 October 2007 Urbana, Illinois, USA, pp.172–186. Available from: http://dx.doi.org/10.1007/978-3-540-85261-2_12.

Dally, B., 2011. Power, programmability, and granularity: the challenges of exascale computing. *25th IEEE international symposium on parallel and distributed processing, IPDPS 2011, conference proceedings* [Online], 16–20 May 2011 Anchorage, Alaska, USA. IEEE, p.878. Available from: http://dx.doi.org/10.1109/IPDPS.2011.420.

Damaraju, S., Varghese, G., Jahagirdar, S., Khondker, T., Milstrey, R., Sarkar, S., Siers, S., Stolero, I. and Subbiah, A., 2012. A 22nm IA multi-CPU and GPU system-on-chip. *2012 IEEE international solid-state circuits conference, ISSCC 2012* [Online], 19–23 February 2012 San Francisco, CA, USA. IEEE, pp.56–57. Available from: http://dx.doi.org/10.1109/ISSCC.2012.6176876.

Damas, L. and Milner, R., 1982. Principal type-schemes for functional programs. *Proceedings of the 9th acm sigplan-sigact symposium on principles of programming languages, popl 1982* [Online]. Albuquerque, New Mexico: ACM, pp.207–212. Available from: http://dx.doi.org/10.1145/582153.582176.

Dammertz, H., Hanika, J. and Keller, A., 2008. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Comput. Graph. Forum* [Online], 27(4), pp.1225–1233. Available from: http://dx.doi.org/10.1111/j.1467-8659.2008.01261.x.

Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V. and Vetter, J.S., 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In: D.R. Kaeli and M. Leeser, eds. *Proceedings of 3rd workshop on general purpose processing on graphics processing units, GPGPU 2010* [Online], 14 March 2010 Pittsburgh, Pennsylvania, USA. Vol. 425, ACM International Conference Proceeding Series. ACM, pp.63–74. Available from: http://dx.doi.org/10.1145/1735688.1735702.

Danowitz, A., Kelley, K., Mao, J., Stevenson, J.P. and Horowitz, M., 2012. CPU DB: recording microprocessor history. *Commun. ACM* [Online], 55(4), pp.55–63. Available from: http://dx.doi.org/10.1145/2133806.2133822.

Deering, M., Winner, S., Schediwy, B., Duffy, C. and Hunt, N., 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In: R.J. Beach, ed. *Proceedings of the 15th annual conference on computer graphics and interactive techniques, SIGGRAPH 1988* [Online], 1–5 August 1988 Atlanta, Georgia, USA. ACM, pp.21–30. Available from: http://dx.doi.org/10.1145/54852.378468.

Demidov, D., 2012. *VexCL*. Accessed: 4 February 2015. Available from: http://ddemidov.github.io/vexcl/.

Demidov, D., Ahnert, K., Rupp, K. and Gottschling, P., 2013. Programming CUDA and opencl: A case study using modern C++ libraries. *SIAM J. Scientific Computing* [Online], 35(5). Available from: http://dx.doi.org/10.1137/120903683.

Dennard, R.H., Gaensslen, F.H., Rideout, V.L., Bassous, E. and LeBlanc, A.R., 1974. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), pp.256–268.

Dolbeau, R., Bihan, S. and Bodin, F., 2007. HMPP: A hybrid multi-core parallel programming environment. *Workshop on general purpose processing on graphics processing units, GPGPU 2007*. Vol. 28.

Donaldson, A.F., Dolinsky, U., Richards, A. and Russell, G., 2010. Automatic offloading of C++ for the cell BE processor: A case study using Offload. In: L. Barolli, F. Xhafa, S. Vitabile and H. Hsu, eds. *CISIS 2010, the fourth international conference on complex, intelligent and software intensive systems* [Online], 15–18 February 2010 Krakow, Poland. IEEE Computer Society, pp.901–906. Available from: http://dx.doi.org/10.1109/CISIS.2010.147.

Doumoulakis, A., Keryell, R. and O'Brien, K., 2017. SYCL C++ and OpenCL interoperability experimentation with triSYCL. *Proceedings of the 5th international workshop on OpenCL* [Online], IWOCL 2017. Toronto, Canada: ACM, 31:1–31:8. Available from: http://dx.doi.org/10.1145/3078155.3078188.

Doyle, M.J., Fowler, C. and Manzke, M., 2012. Hardware accelerated construction of SAH-based bounding volume hierarchies for interactive ray tracing. In: M. Garland, R. Wang, S.N. Spencer, M. Gopi and S. Yoon, eds. *Symposium on interactive 3d graphics and games, I3D 2012* [Online], 9–11 March 2012 Costa Mesa, California, USA. ACM, p.209. Available from: http://dx.doi.org/10.1145/2159616.2159655.

Duran, A. and Klemm, M., 2012. The Intel® many integrated core architecture. *International conference on high performance computing and simulation, HPCS 2012*. IEEE, pp.365–366.

Dütsch, F., Djelassi, K., Haidl, M. and Gorlatch, S., 2014. HLSF: A high-level; C++-based framework for stencil computations on accelerators. *Proceedings of the second workshop on optimizing stencil computations* [Online], WOSC '14. Portland, Oregon, USA: ACM, pp.41–4. Available from: `http://dx.doi.org/10.1145/2686745.2686751`.

Edwards, H.C., Sunderland, D., Porter, V., Amsler, C. and Mish, S., 2012. Manycore performance-portability: kokkos multidimensional array library. *Scientific Programming* [Online], 20(2), pp.89–114. Available from: `http://dx.doi.org/10.3233/SPR-2012-0343`.

Edwards, H.C. and Trott, C.R., 2013. Kokkos: enabling performance portability across manycore architectures. *2013 extreme scaling workshop, xsw 2013*. IEEE, pp.18–24.

Eichenberger, A.E., O'Brien, K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M., Archambault, R., Gao, Y. and Koo, R., 2006. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine$^{tm}$ architecture. *IBM Systems Journal* [Online], 45(1), pp.59–84. Available from: `http://dx.doi.org/10.1147/sj.451.0059`.

Ernst, M. and Greiner, G., 2008. Multi bounding volume hierarchies. *2008 IEEE symposium on interactive ray tracing, RT 2008*. IEEE, pp.35–40.

Falch, T.L. and Elster, A.C., 2016. ImageCL: An image processing language for performance portability on heterogeneous systems. *CoRR* [Online], abs/1605.06399. Available from: `http://arxiv.org/abs/1605.06399`.

Filipovic, J. and Benkner, S., 2015. OpenCL kernel fusion for GPU, Xeon Phi and CPU. *27th international symposium on computer architecture and high performance computing, SBAC-PAD 2015* [Online], 17–21 October 2015 Florianópolis, Brazil. IEEE Computer Society, pp.98–105. Available from: `http://dx.doi.org/10.1109/SBAC-PAD.2015.29`.

Filipovic, J., Madzin, M., Fousek, J. and Matyska, L., 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* [Online], 71(10), pp.3934–3957. Available from: `http://dx.doi.org/10.1007/s11227-015-1483-z`.

Foley, T. and Sugerman, J., 2005. KD-tree acceleration structures for a GPU raytracer. In: M. Meißner and B. Schneider, eds. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware 2005* [Online], 30–31 July 2005 Los Angeles, California, USA. Eurographics Association, pp.15–22. Available from: http://dx.doi.org/10.2312/EGGH/EGGH05/015-022.

Fousek, J., Filipovic, J. and Madzin, M., 2011. Automatic fusions of CUDA-GPU kernels for parallel map. *SIGARCH Computer Architecture News* [Online], 39(4), pp.98–99. Available from: http://dx.doi.org/10.1145/2082156.2082183.

Fuetterling, V., Lojewski, C., Pfreundt, F.-J. and Ebert, A., 2015. Efficient ray tracing kernels for modern cpu architectures. *Journal of Computer Graphics Techniques Vol*, 4(4).

Gaster, B.R., Hower, D. and Howes, L.W., 2015. HRF-relaxed: adapting HRF to the complexities of industrial heterogeneous memory models. *TACO* [Online], 12(1), 7:1–7:26. Available from: http://dx.doi.org/10.1145/2701618.

Gaster, B.R. and Howes, L.W., 2013. OpenCL C++. In: J. Cavazos, X. Gong and D.R. Kaeli, eds. *Proceedings of the 6th workshop on general purpose processor using graphics processing units, GPGPU-6* [Online], 16 March 2013 Houston, Texas, USA. ACM, pp.86–95. Available from: http://dx.doi.org/10.1145/2458523.2458532.

Georgiev, I. and Slusallek, P., 2008. RTfact: generic concepts for flexible and high performance ray tracing. *2008 IEEE symposium on interactive ray tracing, RT 2008*. IEEE, pp.115–122.

Glassner, A.S., 1984. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications* [Online], 4(10), pp.15–24. Available from: http://dx.doi.org/10.1109/MCG.1984.6429331.

Glossner, J., 2016. Heterogeneous systems era. *29th international conference on architecture of computing systems, ARCS 2016*.

Goli, M., 2016. VisionCPP: A SYCL-based computer vision framework. *Proceedings of the 4th international workshop on OpenCL* [Online], Vienna, Austria, IWOCL 2016. New York, NY, USA: ACM, 6:1–6:4. Available from: http://dx.doi.org/10.1145/2909437.2909444.

Goli, M., Iwanski, L. and Richards, A., 2017. Accelerated machine learning using TensorFlow and SYCL on OpenCL devices. In: S. McIntosh-Smith and B. Bergen, eds. *Proceedings of the 5th international workshop on OpenCL, IWOCL 2017* [Online], 16–18 May 2017 Toronto, Canada. ACM, 8:1–8:4. Available from: http://dx.doi.org/10.1145/3078155.3078160.

Green 500, 2015. *The green500 list - november 2015* [Online]. Accessed: 6 June 2016. Available from: http://www.green500.org/lists/green201511.

Guennebaud, G., Jacob, B. et al., 2010. *Eigen v3* [Online]. Available from: http://eigen.tuxfamily.org.

Haidl, M. and Gorlatch, S., 2014. PACXX: towards a unified programming model for programming accelerators using C++14. In: H. Finkel and J.R. Hammond, eds. *Proceedings of the 2014 LLVM compiler infrastructure in HPC, LLVM 2014* [Online], 17 November 2014 New Orleans, Louisiana, USA. ACM, pp.1–11. Available from: http://dl.acm.org/citation.cfm?id=2688363.

Haidl, M., Steuwer, M., Humernbrum, T. and Gorlatch, S., 2016. Multi-stage programming for GPUs in C++ using PACXX. In: D.R. Kaeli and J. Cavazos, eds. *Proceedings of the 9th annual workshop on general purpose processing using graphics processing unit, GPGPU@PPoPP 2016* [Online], 12–16 March 2016 Barcelona, Spain. ACM, pp.32–41. Available from: http://dx.doi.org/10.1145/2884045.2884049.

Han, T.D. and Abdelrahman, T.S., 2009. *HI*CUDA: a high-level directive-based language for GPU programming. In: D.R. Kaeli and M. Leeser, eds. *Proceedings of 2nd workshop on general purpose processing on graphics processing units, GPGPU 2009* [Online], 8 March 2009 Washington, D.C., USA. Vol. 383, ACM International Conference Proceeding Series. ACM, pp.52–61. Available from: http://dx.doi.org/10.1145/1513895.1513902.

Han, T.D. and Abdelrahman, T.S., 2011. Hicuda: high-level GPGPU programming. *IEEE Trans. Parallel Distrib. Syst.* [Online], 22(1), pp.78–90. Available from: http://dx.doi.org/10.1109/TPDS.2010.62.

Hapala, M., Davidovic, T., Wald, I., Havran, V. and Slusallek, P., 2011. Efficient stackless BVH traversal for ray tracing. In: T. Nishita and S.N. Spencer, eds. *Spring conference on computer graphics, SCCG 2011* [Online], 28–30 April 2011 Vininé, Slovakia. ACM, pp.7–12. Available from: http://dx.doi.org/10.1145/2461217.2461219.

Havran, V., Bittner, J. and Zára, J., 1998. Ray tracing with rope trees. *14th spring conference on computer graphics*, pp.130–140.

Hindley, R., 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146, pp.29–60.

Hines, S., Liao, S.-w., Sams, J. and Sakhartchouk, A., 2011. Android renderscript.

Hoberock, J., Kohlhoff, M.G.C., Mysen, C., Edwards, C., Brown, G., Contributors, O., Heller, T., Howes, L., Lelbach, B., Kaiser, H., Lelbach, B., Nishanov, G., Rodgers, T., Hollman, D. and Wong, M., 2017. *A unified executors proposal for C++* [Online]. ISO C++ Study Group 1 - Concurrency and Parallelism. Available from: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0443r1.html`.

Hornung, R., Keasler, J. et al., 2014. The RAJA portability layer: overview and status. *Lawrence Livermore National Laboratory, Livermore, USA*.

Hower, D.R., Hechtman, B.A., Beckmann, B.M., Gaster, B.R., Hill, M.D., Reinhardt, S.K. and Wood, D.A., 2014. Heterogeneous-race-free memory models. In: R. Balasubramonian, A. Davis and S.V. Adve, eds. *Architectural support for programming languages and operating systems, ASPLOS 2014* [Online], 1–5 March 2014 Salt Lake City, Utah, USA. ACM, pp.427–440. Available from: `http://dx.doi.org/10.1145/2541940.2541981`.

Howes, L.W., Lokhmotov, A., Donaldson, A.F. and Kelly, P.H.J., 2009a. Deriving efficient data movement from decoupled access/execute specifications. In: A. Seznec, J.S. Emer, M.F.P. O'Boyle, M. Martonosi and T. Ungerer, eds. *High performance embedded architectures and compilers, fourth international conference, HiPEAC 2009, proceedings* [Online], 25–28 January 2009 Paphos, Cyprus. Vol. 5409, Lecture Notes in Computer Science. Springer, pp.168–182. Available from: `http://dx.doi.org/10.1007/978-3-540-92990-1_14`.

Howes, L.W., Lokhmotov, A., Donaldson, A.F. and Kelly, P.H.J., 2009b. Towards metaprogramming for parallel systems on a chip. In: H. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa and A. Streit, eds. *Euro-par 2009 - parallel processing workshops, HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, 2009, revised selected papers* [Online], 25–28 August 2009 Delft, The Netherlands. Vol. 6043, Lecture Notes in Computer Science. Springer, pp.36–45. Available from: `http://dx.doi.org/10.1007/978-3-642-14122-5_7`.

HSA Foundation, 2015a. *HSA platform system architecture specification* [Online]. (V.1.0). Available from: http://www.hsafoundation.com/?ddownload=4944.

HSA Foundation, 2015b. *HSA programmers reference manual* [Online]. (V.1.0). Available from: http://www.hsafoundation.com/?ddownload=4945.

HSA Foundation, 2015c. *HSA runtime programmers reference manual* [Online]. (V.1.0). Available from: http://www.hsafoundation.com/?ddownload=4946.

HSA Foundation, 2016a. *HSA platform system architecture specification* [Online]. (V.1.1). Available from: http://www.hsafoundation.com/?ddownload=5114.

HSA Foundation, 2016b. *HSA programmers reference manual* [Online]. (V.1.1). Available from: http://www.hsafoundation.com/?ddownload=5577.

Immel, D.S., Cohen, M.F. and Greenberg, D.P., 1986. A radiosity method for non-diffuse environments. In: D.C. Evans and R.J. Athay, eds. *Proceedings of the 13th annual conference on computer graphics and interactive techniques, SIGGRAPH 1986* [Online], 18–22 August 1986 Dallas, Texas, USA. ACM, pp.133–142. Available from: http://dx.doi.org/10.1145/15922.15901.

Intel, 2016. *Threading building blocks* [Online]. Available from: https://www.threadingbuildingblocks.org/.

Intel. *Intel® FPGA SDK for open computing language* [Online]. Accessed: 3 June 2017. Available from: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

Intel. *Intel® SDK for OpenCL™ applications* [Online]. Accessed: 3 June 2017. Available from: https://software.intel.com/en-us/intel-opencl.

International Business Machines, 2008. *Getting started with XL C/C++* [Online]. (V.10.1). Accessed: 24 May 2016. Available from: http://www-01.ibm.com/support/docview.wss?uid=swg27014219&aid=1.

ISO/IEC, 1999. *ISO/IEC 9899:1999 - Programming languages - C*. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission, (Standard).

ISO/IEC, 2006. *ISO/IEC TR 18037:2006 - Programming languages - C - Extensions to support embedded processors* [Online]. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission, (Draft Technical Report), pp.1–97. Available from: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf.

ISO/IEC, 2014. *ISO/IEC 14882:2014(E) - Information technology - Programming languages - C*. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission, (Standard), pp.1–1358.

ISO/IEC, 2015. *ISO/IEC ISO/IEC JTC1 SC22 WG21 N4507 - Programming Languages - Technical Specification for C++ Extensions for Parallelism* [Online]. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission, (tech. rep.), pp.1–23. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf.

Itseez, 2015. *Open source computer vision library*. Available from: https://github.com/itseez/opencv.

Jääskeläinen, P., de La Lama, C.S., Schnetter, E., Raiskila, K., Takala, J. and Berg, H., 2014. Pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming* [Online]. Available from: http://dx.doi.org/10.1007/s10766-014-0320-y.

Jambor, M., 2015. *HSA-OpenMP-GCC-AMD* [Online]. Available from: https://github.com/HSAFoundation/HSA-OpenMP-GCC-AMD/wiki.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T., 2014. Caffe: convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T.V., Gottipati, R., Gulland, W., Hagmann, R., Ho, R.C., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing,

A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E. and Yoon, D.H., 2017. In-datacenter performance analysis of a tensor processing unit. *CoRR* [Online], abs/1704.04760. Available from: http://arxiv.org/abs/1704.04760.

Kaiser, H., Adelstein-Lelbach, B., Heller, T., Bergé, A., Biddiscombe, J., Bikineev, A., Mercer, G., Schäfer, A., Habraken, J., Serio, A., Anderson, M., Stumpf, M., Bourgeois, D., Grubel, P., Brandt, S.R., Copik, M., Amatya, V., Huck, K., Viklund, L., Khatami, Z., Bacharwar, D., Yang, S., Schnetter, E., Bcorde5, Brodowicz, M., Bibek, atrantan, Troska, L., Byerly, Z. and Upadhyay, S., 2016. *Hpx: HPX V0.9.99: A general purpose C++ runtime system for parallel and distributed applications of any scale* [Online]. Available from: http://dx.doi.org/10.5281/zenodo.58027.

Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A. and Fey, D., 2014. HPX: A task based programming model in a global address space. In: A.D. Malony and J.R. Hammond, eds. *Proceedings of the 8th international conference on partitioned global address space programming models, PGAS 2014* [Online], 6–10 October 2014 Eugene, Oregon, USA. ACM, 6:1–6:11. Available from: http://dx.doi.org/10.1145/2676870.2676883.

Kajiya, J.T., 1986. The rendering equation. In: D.C. Evans and R.J. Athay, eds. *Proceedings of the 13th annual conference on computer graphics and interactive techniques, SIGGRAPH 1986* [Online], 18–22 August 1986 Dallas, Texas, USA. ACM, pp.143–150. Available from: http://dx.doi.org/10.1145/15922.15902.

Kay, T.L. and Kajiya, J.T., 1986. Ray tracing complex scenes. In: D.C. Evans and R.J. Athay, eds. *Proceedings of the 13th annual conference on computer graphics and interactive techniques, SIGGRAPH 1986* [Online], 18–22 August 1986 Dallas, Texas, USA. ACM, pp.269–278. Available from: http://dx.doi.org/10.1145/15922.15916.

Keller, A., Karras, T., Wald, I., Aila, T., Laine, S., Bikker, J., Gribble, C.P., Lee, W. and McCombe, J., 2013. Ray tracing is the future and ever will be.. *International conference on computer graphics and interactive techniques, SIGGRAPH 2013, courses* [Online], 21–25 July 2013 Anaheim, California, USA. ACM, 9:1–9:7. Available from: http://dx.doi.org/10.1145/2504435.2504444.

Keryell, R., 2015. *triSYCL*. Available from: https://github.com/triSYCL/triSYCL.

Khronos OpenCL Working Group, 2008. *The OpenCL specification* [Online]. A. Munshi, ed. (V.1.0). Khronos Group. Available from: https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf.

Khronos OpenCL Working Group, 2012. *The OpenCL specification* [Online]. A. Munshi, ed. (V.1.2, rev 19). Khronos Group. Available from: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

Khronos OpenCL Working Group, 2013. *The OpenCL C++ wrapper API* [Online]. B.R. Gaster and L. Howes, eds. (V.1.2.6). Khronos Group. Available from: https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf.

Khronos OpenCL Working Group, 2016. *The OpenCL C specification* [Online]. A. Munshi, L. Howes and B. Sochacki, eds. Khronos Group. Available from: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf.

Khronos OpenCL Working Group, 2017. *The OpenCL C++ 1.0 specification* [Online]. Bartosz Sochacki, ed. (V.1.0, rev 24). Khronos Group. Available from: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2-cplusplus.pdf.

Khronos OpenCL Working Group – SPIR subgroup, 2014. *The SPIR Specification* [Online]. B. Ouriel, ed. (V.1.2). Khronos Group. Available from: https://www.khronos.org/registry/SPIR/specs/spir_spec-1.2.pdf.

Khronos OpenCL Working Group – SYCL subgroup, 2015. *SYCL specification* [Online]. L. Howes and M. Rovatsou, eds. (V.1.2). Khronos Group. Available from: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf.

Khronos OpenGL ES Working Group, 2007. *OpenGL ES common profile specification 2.0* [Online]. A. Munshi and J. Leech, eds. Available from: https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_full_spec_2.0.pdf.

Khronos OpenGL Working Group, 2016a. *The OpenGL® graphics system: a specification* [Online]. M. Segal and K. Akeley, eds. Khronos Group. Available from: https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf.

Khronos OpenGL Working Group, 2016b. *The OpenGL® shading language* [Online]. John Kessenich, ed. Khronos Group. Available from: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf.

Khronos SPIR-V Working Group, 2016. *SPIR-V specification* [Online]. J. Kessenich and B. Ouriel, eds. Khronos Group. Available from: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf.

Khronos Vulkan Working Group, 2016. *Vulkan - a specification* [Online]. (V.1.0.32). Khronos Group. Available from: https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf.

Kiemele, L., Berg, C., Gulliver, A. and Coady, Y., 2013. Kfusion: optimizing data flow without compromising modularity. *Proceedings of the 12th annual international conference on aspect-oriented software development, aosd 2013* [Online], Fukuoka, Japan. New York, NY, USA: ACM, pp.25–36. Available from: http://dx.doi.org/10.1145/2451436.2451440.

Kindlmann, G.L., Chiw, C., Seltzer, N., Samuels, L. and Reppy, J.H., 2016. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Trans. Vis. Comput. Graph.* [Online], 22(1), pp.867–876. Available from: http://dx.doi.org/10.1109/TVCG.2015.2467449.

Kjolstad, F., Kamil, S., Ragan-Kelley, J., Levin, D.I.W., Sueda, S., Chen, D., Vouga, E., Kaufman, D.M., Kanwar, G., Matusik, W. and Amarasinghe, S.P., 2016. Simit: A language for physical simulation. *ACM Trans. Graph.* [Online], 35(2), p.20. Available from: http://dx.doi.org/10.1145/2866569.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. and Fasih, A., 2012. Pycuda and pyopencl: A scripting-based approach to GPU run-time code generation. *Parallel Computing* [Online], 38(3), pp.157–174. Available from: http://dx.doi.org/10.1016/j.parco.2011.09.001.

Kopta, D., Shkurko, K., Spjut, J.B., Brunvand, E. and Davis, A., 2015. Memory considerations for low energy ray tracing. *Comput. Graph. Forum* [Online], 34(1), pp.47–59. Available from: http://dx.doi.org/10.1111/cgf.12458.

Krieger, P., 2016. *Vulkano*. Available from: https://github.com/vulkano-rs/vulkano.

Krokstad, A., Strom, S. and Sørsdal, S., 1968. Calculating the acoustical room response by the use of a ray tracing technique. *Journal of Sound and Vibration*, 8(1), pp.118–125.

Kulowski, A., 1985. Algorithmic representation of the ray tracing technique. *Applied Acoustics*, 18(6), pp.449–469.

Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P. and Tullsen, D.M., 2003. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. *Proceedings of the 36th annual international symposium on microarchitecture* [Online], 3–5 December 2003 San Diego, California, USA. ACM/IEEE Computer Society, pp.81–92. Available from: http://dx.doi.org/10.1109/MICRO.2003.1253185.

Lafortune, E.P. and Willems, Y.D., 1993. Bi-directional path tracing. *Proceedings of third international conference on computational graphics and visualization techniques, Compugraphics 1993*, pp.145–153.

Laine, S., 2010. Restart trail for stackless bvh traversal. *Proceedings of the conference on high performance graphics, HPG 2010* [Online], Saarbrücken, Germany. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, pp.107–111. Available from: http://dl.acm.org/citation.cfm?id=1921479.1921496.

Laine, S., Karras, T. and Aila, T., 2013. Megakernels considered harmful: wavefront path tracing on gpus. In: K. Fatahalian, C. Theobalt and J. Lehtinen, eds. *High-performance graphics, HPG 2013, proceedings* [Online], 19–21 July 2013 Anaheim, California, USA. ACM, pp.137–144. Available from: http://dx.doi.org/10.1145/2492045.2492060.

Lam, S.K., Pitrou, A. and Seibert, S., 2015. Numba: a LLVM-based python JIT compiler. In: H. Finkel, ed. *Proceedings of the second workshop on the LLVM compiler infrastructure in hpc, LLVM 2015* [Online], 15 November 2015 Austin, Texas, USA. ACM, 7:1–7:6. Available from: http://dx.doi.org/10.1145/2833157.2833162.

Landaverde, R., Zhang, T., Coskun, A.K. and Herbordt, M.C., 2014. An investigation of unified memory access performance in CUDA. *IEEE high performance extreme computing conference, HPEC 2014* [Online], 9–11 September 2014 Waltham, Massachusetts, USA. IEEE, pp.1–6. Available from: http://dx.doi.org/10.1109/HPEC.2014.7040988.

Lattner, C. et al., 2007. *A C language family frontend for LLVM*. Accessed: 24 November 2015. Available from: http://clang.llvm.org/.

Lattner, C. and Adve, V.S., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *2nd IEEE / ACM international symposium on code generation and optimization, CGO 2004* [Online], 20–24 March 2004 San Jose, CA, USA. IEEE Computer Society, pp.75–88. Available from: http://dx.doi.org/10.1109/CGO.2004.1281665.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D.P. and Manocha, D., 2009. Fast BVH construction on gpus. *Comput. Graph. Forum* [Online], 28(2), pp.375–384. Available from: http://dx.doi.org/10.1111/j.1467-8659.2009.01377.x.

Lee, S. and Eigenmann, R., 2010. OpenMPC: extended OpenMP programming and tuning for GPUs. *Conference on high performance computing networking, storage and analysis, SC 2010* [Online], 13–19 November 2010 New Orleans, Louisiana, USA. IEEE, pp.1–11. Available from: http://dx.doi.org/10.1109/SC.2010.36.

Lee, W., Hwang, S.J., Shin, Y., Yoo, J. and Ryu, S., 2015. An efficient hybrid ray tracing and rasterizer architecture for mobile GPU. *SIGGRAPH Asia 2015 mobile graphics and interactive applications* [Online], 2–6 November 2015 Kobe, Japan. ACM, 2:1–2:4. Available from: http://dx.doi.org/10.1145/2818427.2818442.

Lee, W., Shin, Y., Lee, J.D., Hwang, S.J., Ryu, S. and Kim, J., 2014. An energy efficient hardware multithreading scheme for mobile ray tracing. *SIGGRAPH Asia 2014 mobile graphics and interactive applications* [Online], 3–6 December 2014 Shenzhen, China. ACM, 1:1–1:3. Available from: http://dx.doi.org/10.1145/2669062.2669079.

Leißa, R., Boesche, K., Hack, S., Membarth, R. and Slusallek, P., 2015. Shallow embedding of dsls via online partial evaluation. In: C. Kästner and A.S. Gokhale, eds. *Proceedings of the 2015 ACM SIGPLAN international conference on generative programming: concepts and experiences, GPCE 2015* [Online], 26–27 October 2015 Pittsburgh, Pennsylvania, USA. ACM, pp.11–20. Available from: http://dx.doi.org/10.1145/2814204.2814208.

Leißa, R., Köster, M. and Hack, S., 2015. A graph-based higher-order intermediate representation. In: K. Olukotun, A. Smith, R. Hundt and J. Mars, eds. *Proceedings of the 13th annual IEEE/ACM international symposium on code generation and optimization, CGO 2015* [Online], 7–11 February 2015 San Francisco, California, USA. IEEE Computer Society, pp.202–212. Available from: http://dx.doi.org/10.1109/CGO.2015.7054200.

Lepley, T., Paulin, P.G. and Flamand, E., 2013. A novel compilation approach for image processing graphs on a many-core platform with explicitly managed memory. *International conference on compilers, architecture and synthesis for embedded systems, CASES 2013* [Online], 29 September–4 October 2013 Montreal, Quebec, Canada. IEEE, 6:1–6:10. Available from: http://dx.doi.org/10.1109/CASES.2013.6662510.

Li, K. and Hudak, P., 1986. Memory coherence in shared virtual memory systems. In: J.Y. Halpern, ed. *Proceedings of the fifth annual ACM symposium on principles of distributed computing* [Online], 11–13 August 1986 Calgary, Alberta, Canada. ACM, pp.229–239. Available from: http://dx.doi.org/10.1145/10590.10610.

Lutz, K. *Boost.Compute*. Accessed: 29 November 2015. Available from: https://boostorg.github.io/compute/.

MacDonald, J.D. and Booth, K.S., 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* [Online], 6(3), pp.153–166. Available from: http://dx.doi.org/10.1007/BF01911006.

Malcolm, J., Yalamanchili, P., McClanahan, C., Venugopalakrishnan, V., Patel, K. and Melonakos, J., 2012. Arrayfire: a GPU acceleration platform. *Modeling and simulation for defense systems and applications VII* [Online]. Vol. 8403. SPIE, p.6. Available from: http://dx.doi.org/10.1117/12.921122.

Mansson, E., Munkberg, J. and Akenine-Möller, T., 2007. Deep coherent ray tracing. *2007 IEEE symposium on interactive ray tracing, RT 2007*. IEEE, pp.79–85.

Mark, W.R., Glanville, R.S., Akeley, K. and Kilgard, M.J., 2003. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.* [Online], 22(3), pp.896–907. Available from: http://dx.doi.org/10.1145/882262.882362.

Märtin, C., 2014. Post-Dennard scaling and the final years of Moore's law consequences for the evolution of multicore-architectures. *Informatik und Interaktive Systeme*.

Matsuzaki, K. and Emoto, K., 2009. Implementing fusion-equipped parallel skeletons by expression templates. In: M.T. Morazán and S. Scholz, eds. *Implementation and application of functional languages - 21st international symposium, IFL 2009, revised selected papers* [Online], 23–25 September 2009 South Orange, New Jersey, USA. Vol. 6041, Lecture Notes in Computer Science. Springer, pp.72–89. Available from: http://dx.doi.org/10.1007/978-3-642-16478-1_5.

McCalpin, J.D., 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*.

McCool, M.D., Qin, Z. and Popa, T.S., 2002. Shader metaprogramming. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware, HWWS 2002* [Online], Saarbrücken, Germany. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, pp.57–68. Available from: http://dl.acm.org/citation.cfm?id=569046.569055.

McCool, M., Du Toit, S., Popa, T., Chan, B. and Moule, K., 2004. Shader algebra. *ACM Trans. Graph.* [Online], 23(3) (), pp.787–795. Available from: http://dx.doi.org/10.1145/1015706.1015801.

Membarth, R., Reiche, O., Hannig, F., Teich, J., Körner, M. and Eckert, W., 2016. Hipa$^{cc}$: A domain-specific language and compiler for image processing. *IEEE Trans. Parallel Distrib. Syst.* [Online], 27(1), pp.210–224. Available from: http://dx.doi.org/10.1109/TPDS.2015.2394802.

Menon, J., De Kruijf, M. and Sankaralingam, K., 2012. iGPU: exception support and speculative execution on GPUs. *SIGARCH Comput. Archit. News* [Online], 40(3) (), pp.72–83. Available from: http://dx.doi.org/10.1145/2366231.2337168.

Microsoft Corporation, 2013. *C++ AMP: language and programming model*. (V.1.2).

Milner, R., 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3), pp.348–375.

Mollick, E., 2006. Establishing Moore's law. *IEEE Annals of the History of Computing* [Online], 28(3), pp.62–75. Available from: http://dx.doi.org/10.1109/MAHC.2006.45.

Monteyne, M., 2008. *RapidMind multi-core development platform*.

Mortensen, J., Khanna, P., Yu, I. and Slater, M., 2007. A visibility field for ray tracing. *4th international conference on computer graphics, imaging and visualization, CGIV 2007* [Online], 14–16 August 2007 Bangkok, Thailand. IEEE Computer Society, pp.54–61. Available from: http://dx.doi.org/10.1109/CGIV.2007.14.

Mullapudi, R.T., Vasista, V. and Bondhugula, U., 2015. PolyMage: automatic optimization for image processing pipelines. In: Ö. Özturk, K. Ebcioglu and S. Dwarkadas, eds. *Proceedings of the twentieth international conference on architectural support for programming languages and operating systems, ASPLOS 2015* [Online], 14–18 March 2015 Istanbul, Turkey. ACM, pp.429–443. Available from: http://dx.doi.org/10.1145/2694344.2694364.

Nah, J.-H., Kang, Y.-S., Lee, K.-J., Lee, S.-J., Han, T.-D. and Yang, S.-B., 2010. MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices. *ACM SIGGRAPH Asia 2010 sketches*. ACM, p.50.

Nah, J., Kwon, H., Kim, D., Jeong, C., Park, J., Han, T., Manocha, D. and Park, W., 2014. Raycore: A ray-tracing hardware architecture for mobile devices. *ACM Trans. Graph.* [Online], 33(5), 162:1–162:15. Available from: http://dx.doi.org/10.1145/2629634.

Newburn, C.J., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chinchilla, F. and McGuire, R., 2013. Offload compiler runtime for the Intel Xeon Phi coprocessor. In: J.M. Kunkel, T. Ludwig and H.W. Meuer, eds. *Supercomputing - 28th international supercomputing conference, ISC 2013, proceedings* [Online], 16–20 June 2013 Leipzig, Germany. Vol. 7905, Lecture Notes in Computer Science. Springer, pp.239–254. Available from: http://dx.doi.org/10.1007/978-3-642-38750-0_18.

Ni, T., 2009. Direct Compute: bring GPU computing to the mainstream. *Gpu technology conference*, p.23.

Nicholas, J., 2015. *Heterogeneous Systems Architecture: coming soon to a platform near you* [Online]. Invited Talk at Linley Processor Conference 2015. Available from: http://www.hsafoundation.com/wp-content/uploads/2013/11/HSA_Coming_Soon_To_A_Platform_Near_You.pdf.

NVIDIA Corporation, 2007. *CUDA programming guide*. (V.1.0).

NVIDIA Corporation, 2011. *NVIDIA Performance Primitives* [Online]. Accessed: 4 February 2015. Available from: http://developer.nvidia.com/npp.

Oneppo, M., 2007. HLSL shader model 4.0. In: S. McMains and P. Sloan, eds. *34th international conference on computer graphics and interactive techniques, SIGGRAPH 2007, courses* [Online], 5–9 August 2007 San Diego, California, USA. ACM, pp.112–152. Available from: http://dx.doi.org/10.1145/1281500.1281576.

OpenACC Working Group and others, 2011. *The OpenACC application programming interface* [Online]. Accessed: 9 June 2016. Available from: http://www.openacc.org/.

OpenMP ARB, 2015. *OpenMP application programming interface*. (V.4.5). Accessed: 9 June 2016. Available from: http://www.openmp.org/mp-documents/openmp-4.5.pdf.

Pajot, A., Barthe, L., Paulin, M. and Poulin, P., 2011. Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering. *Comput. Graph. Forum* [Online], 30(2), pp.315–324. Available from: http://dx.doi.org/10.1111/j.1467-8659.2011.01863.x.

Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D.P., McAllister, D.K., McGuire, M., Morley, R.K., Robison, A. and Stich, M., 2010. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.* [Online], 29(4), 66:1–66:13. Available from: http://dx.doi.org/10.1145/1833351.1778803.

Potter, R., 2015. *Models for high level languages on HSA*. HSA Foundation Working Group. Cambridge, UK.

Potter, R., 2016a. *A C++ programming model for Heterogeneous System Architecture*. HSA Foundation Working Group. Edinburgh, UK.

Potter, R., 2016b. *A C++ programming model for Heterogeneous System Architecture*. SG14 - ISO C++ Study Group on Games Development and Low-Latency.

Potter, R., 2016c. *Generating efficient GPU kernels using expression templates*. HSA Foundation Working Group. Edinburgh, UK.

Purcell, T.J., Buck, I., Mark, W.R. and Hanrahan, P., 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.* [Online], 21(3), pp.703–712. Available from: http://dx.doi.org/10.1145/566654.566640.

Qualcomm, 2017. *Snapdragon mobile processors and chipsets* [Online]. Accessed: 3 April 2017. Available from: https://www.qualcomm.com/products/snapdragon.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. and Amarasinghe, S.P., 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: H. Boehm and C. Flanagan, eds. *ACM SIGPLAN conference on programming language design and implementation, PLDI 2013* [Online], 16–19 June 2013 Seattle, Washington, USA. ACM, pp.519–530. Available from: http://dx.doi.org/10.1145/2462156.2462176.

Robison, A., 2011. Domain specific compilation in the NVIDIA optix ray tracing engine. In: M. Carro and J.H. Reppy, eds. *Proceedings of the POPL 2011 workshop on declarative aspects of multicore programming, DAMP 2011* [Online], 23 January 2011 Austin, Texas, USA. ACM, pp.1–2. Available from: http://dx.doi.org/10.1145/1926354.1926356.

Rodgers, G., 2015. *CLOC compiler and sample SDK*. Accessed: 29 November 2015. Available from: https://github.com/HSAFoundation/CLOC.

Rogvall, T., 2009. *OpenCL bindings for Erlang*. Available from: https://github.com/tonyrog/cl/.

Rubin, S.M. and Whitted, T., 1980. A 3-dimensional representation for fast rendering of complex scenes. In: J.J. Thomas, R.A. Ellis and H.Z. Kriloff, eds. *Proceedings of the 7th annual conference on computer graphics and interactive techniques, SIGGRAPH 1980* [Online], 14–18 July 1980 Seattle, Washington, USA. ACM, pp.110–116. Available from: http://dx.doi.org/10.1145/800250.807479.

Rupp, K., Rudolf, F. and Weinbub, J., 2010. ViennaCL - A high level linear algebra library for GPUs and multi-core CPUs. *Intl. workshop on gpus and scientific applications*, pp.51–56.

Sabino, T.L., Andrade, P., Clua, E.W.G., Montenegro, A.A. and Pagliosa, P.A., 2012. A hybrid GPU rasterized and ray traced rendering pipeline for real time rendering of per pixel effects. In: M. Herrlich, R. Malaka and M. Masuch, eds. *Entertainment computing - ICEC 2012 - 11th international conference, ICEC 2012, proceedings* [Online], 26–29 September 2012 Bremen, Germany. Vol. 7522, Lecture Notes in Computer Science. Springer, pp.292–305. Available from: http://dx.doi.org/10.1007/978-3-642-33542-6_25.

Sagan, H., 2012. *Space-filling curves*. Springer Science & Business Media.

Saito, T. and Takahashi, T., 1990. Comprehensible rendering of 3-d shapes. In: F. Baskett, ed. *Proceedings of the 17th annual conference on computer graphics and interactive techniques, SIGGRAPH 1990* [Online], 6 August–10 June 1990 Dallas, Texas, USA. ACM, pp.197–206. Available from: http://dx.doi.org/10.1145/97879.97901.

Sander, B., Stoner, G., Chan, S.-c. and Chung, W.-H., 2015. *HCC: A C++ compiler for heterogeneous computing*. ISO/IEC JTC1 SC22 WG21, (tech. rep.).

Sanderson, C., 2010. *Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments* [Online]. NICTA, (tech. rep.). Available from: http://arma.sourceforge.net/armadillo_nicta_2010.pdf.

Santos, A.L. dos, Teixeira, J.M.X.N., Farias, T.S.M.C. de, Teichrieb, V. and Kelner, J., 2012. Understanding the efficiency of KD-tree ray-traversal techniques over a GPGPU architecture. *International Journal of Parallel Programming* [Online], 40(3), pp.331–352. Available from: http://dx.doi.org/10.1007/s10766-011-0186-1.

Schreiber, W.F., 1970. Wirephoto quality improvement by unsharp masking. *Pattern Recognition* [Online], 2(2), pp.117–120. Available from: http://dx.doi.org/10.1016/0031-3203(70)90007-5.

Sharlet, D., Kunze, A., Junkins, S. and Joshi, D., 2012. Shevlin Park: Implementing C++ AMP with clang/LLVM and OpenCL.

Smistad, E., Bozorgi, M. and Lindseth, F., 2015. FAST: framework for heterogeneous medical image computing and visualization. *Int. J. Computer Assisted Radiology and Surgery* [Online], 10(11), pp.1811–1822. Available from: http://dx.doi.org/10.1007/s11548-015-1158-5.

Sobel, I. and Feldman, G., 1968. *A 3x3 isotropic gradient operator for image processing*.

Sorin, D.J., Hill, M.D. and Wood, D.A., 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), pp.1–212.

Spjut, J., Kopta, D., Brunvand, E. and Davis, A., 2012. A mobile accelerator architecture for ray tracing. *Proceedings of 3rd workshop on SoCs, heterogeneous architectures and workloads SHAW-3*.

Sujeeth, A.K., Lee, H., Brown, K.J., Rompf, T., Chafi, H., Wu, M., Atreya, A.R., Odersky, M. and Olukotun, K., 2011. OptiML: An implicitly parallel domain-specific language for machine learning. In: L. Getoor and T. Scheffer, eds. *Proceedings of the 28th international conference on machine learning, ICML 2011*, 28 June–2 July 2011 Bellevue, Washington, USA. Omnipress, pp.609–616.

Sung, H., Chen, T., Sura, Z. and Islam, T., 2017. Leveraging openmp 4.5 support in CLANG for fortran. In: B.R. de Supinski, S.L. Olivier, C. Terboven, B.M. Chapman and M.S. Müller, eds. *Scaling openmp for exascale performance and portability - 13th international workshop on openmp, IWOMP 2017. proceedings* [Online], 20–22 September 2017 Stony Brook, New York, USA. Vol. 10468, Lecture Notes in Computer Science.

Springer, pp.33–47. Available from: http://dx.doi.org/10.1007/978-3-319-65578-9_3.

Sutter, H., 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3), pp.202–210.

Syncleus, 2016. *Aparapi* [Online]. Available from: http://aparapi.com/.

Taylor, S., 2007. *Optimizing applications for multi-core processors, using the intel integrated performance primitives*. Intel Press.

Texas Instruments. *TI OpenCL vo1.01.xx - TI OpenCL documentation* [Online]. Accessed: 3 June 2017. Available from: http://downloads.ti.com/mctools/esd/docs/opencl/index.html.

Vandevoorde, D. and Josuttis, N.M., 2002. *C++ templates*. Addison-Wesley Longman Publishing Co., Inc.

Veach, E., 1997. *Robust monte carlo methods for light transport simulation*. PhD thesis. Stanford University.

Veldhuizen, T., 1995. Expression templates, C++ report.

Veldhuizen, T.L., 1998. Arrays in Blitz++. In: D. Caromel, R.R. Oldehoeft and M. Tholburn, eds. *Computing in object-oriented parallel environments, second international symposium, ISCOPE 1998, proceedings* [Online], 8–11 December 1998 Santa Fe, New Mexico, USA. Vol. 1505, Lecture Notes in Computer Science. Springer, pp.223–230. Available from: http://dx.doi.org/10.1007/3-540-49372-7_24.

Veldhuizen, T.L., 2000. Blitz++: the library that thinks it is a compiler. In: *Advances in software tools for scientific computing* [Online]. H.P. Langtangen, A.M. Bruaset and E. Quak, eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.57–87. Available from: http://dx.doi.org/10.1007/978-3-642-57172-5_2.

Veldhuizen, T.L. and Jernigan, M.E., 1997. Will C++ be faster than fortran? In: Y. Ishikawa, R.R. Oldehoeft, J. Reynders and M. Tholburn, eds. *Scientific computing in object-oriented parallel environments, ISCOPE 1997, proceedings* [Online], 8–11 December 1997 Marina del Rey, California, USA. Vol. 1343, Lecture Notes in Computer Science. Springer, pp.49–56. Available from: http://dx.doi.org/10.1007/3-540-63827-X_43.

Videau, B., 2013. *OpenCL bindings for Ruby*. Available from: `https://github.com/Nanosim-LIG/opencl-ruby`.

Viitanen, T., Koskela, M., Jääskeläinen, P. and Takala, J., 2016. Multi bounding volume hierarchies for ray tracing pipelines. In: J. Kopf and P.C. Fu, eds. *SIGGRAPH Asia 2016, technical briefs* [Online], 5–8 December 2016 Macao, China. ACM, p.8. Available from: `http://dx.doi.org/10.1145/3005358`.

Wahib, M. and Maruyama, N., 2014. Scalable kernel fusion for memory-bound gpu applications. *Proceedings of the international conference for high performance computing, networking, storage and analysis* [Online], SC '14. New Orleans, Louisana: IEEE Press, pp.191–202. Available from: `http://dx.doi.org/10.1109/SC.2014.21`.

Wald, I., 2007. On fast construction of sah-based bounding volume hierarchies. *2007 IEEE symposium on interactive ray tracing, RT 2007*. IEEE, pp.33–40.

Wald, I., Benthin, C. and Boulos, S., 2008. Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs. *2008 IEEE symposium on interactive ray tracing, RT 2008*. IEEE, pp.49–57.

Wald, I., Johnson, G.P., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Gunther, J. and Navrátil, P.A., 2017. Ospray - A CPU ray tracing framework for scientific visualization. *IEEE Trans. Vis. Comput. Graph.* [Online], 23(1), pp.931–940. Available from: `http://dx.doi.org/10.1109/TVCG.2016.2599041`.

Wald, I., Slusallek, P., Benthin, C. and Wagner, M., 2001. Interactive rendering with coherent ray tracing. *Comput. Graph. Forum* [Online], 20(3), pp.153–165. Available from: `http://dx.doi.org/10.1111/1467-8659.00508`.

Wald, I., Woop, S., Benthin, C., Johnson, G.S. and Ernst, M., 2014. Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.* [Online], 33(4), 143:1–143:8. Available from: `http://dx.doi.org/10.1145/2601097.2601199`.

Wallcraft, A.J., 2002. A comparison of co-array fortran and openmp fortran for SPMD programming. *The Journal of Supercomputing* [Online], 22(3), pp.231–250. Available from: `http://dx.doi.org/10.1023/A:1015322200593`.

Walter, J. and Koch, M., 2002. *The Boost uBLAS library*. Boost. Available from: `http://www.boost.org/doc/libs/1_66_0/libs/numeric/ublas/doc/index.html`.

Wang, G., Lin, Y. and Yi, W., 2010. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In: P. Zhu, L. Wang, F. Xia, H. Chen, I. McLoughlin, S. Tsao, M. Sato, S. Chai and I. King, eds. *2010 IEEE/ACM int'l conference on green computing and communications, greencom 2010, & int'l conference on cyber, physical and social computing, CPSCom 2010* [Online], 18–20 December 2010 Hangzhou, China. IEEE Computer Society, pp.344–350. Available from: http://dx.doi.org/10.1109/GreenCom-CPSCom.2010.102.

Wang, Y. and Cheng, K.(, 2012. Energy and performance characterization of mobile heterogeneous computing. *2012 IEEE workshop on signal processing systems* [Online], 17–19 October 2012 Quebec City, Quebec, Canada. IEEE, pp.312–317. Available from: http://dx.doi.org/10.1109/SiPS.2012.61.

Weinbub, J., Rupp, K. and Selberherr, S., 2011. Towards distributed heterogenous high-performance computing with ViennaCL. In: I. Lirkov, S. Margenov and J. Wasniewski, eds. *Large-scale scientific computing - 8th international conference, LSSC 2011, revised selected papers* [Online], 6–10 June 2011 Sozopol, Bulgaria. Vol. 7116, Lecture Notes in Computer Science. Springer, pp.359–367. Available from: http://dx.doi.org/10.1007/978-3-642-29843-1_41.

Whitted, T., 1979. An improved illumination model for shaded display. In: T.A. DeFanti, B.H. McCormick, B.W. Pollack, N.I. Badler and S.H. Chasen, eds. *Proceedings of the 6th annual conference on computer graphics and interactive techniques, SIGGRAPH 1979* [Online], 8–10 August 1979 Chicago, Illinois, USA. ACM, p.14. Available from: http://dx.doi.org/10.1145/800249.807419.

Wilson, N., 2017. *DCompute: GPGPU with Native D for OpenCL and CUDA* [Online]. Available from: https://dlang.org/blog/2017/07/17/dcompute-gpgpu-with-native-d-for-opencl-and-cuda/.

Wirth, M.A. and Nikitenko, D., 2010. The effect of colour space on image sharpening algorithms. *Canadian conference on computer and robot vision, CRV 2010* [Online], 31 May–2 June 2010 Ottawa, Ontario, Canada. IEEE Computer Society, pp.79–85. Available from: http://dx.doi.org/10.1109/CRV.2010.17.

Wolfe, M., 2010. Implementing the PGI accelerator model. In: D.R. Kaeli and M. Leeser, eds. *Proceedings of 3rd workshop on general purpose processing on graphics processing units, GPGPU 2010* [Online], 14 March 2010 Pittsburgh, Pennsylvania, USA. Vol. 425, ACM International Conference Proceeding Series. ACM, pp.43–50. Available from: http://dx.doi.org/10.1145/1735688.1735697.

Woop, S., Feng, L., Wald, I. and Benthin, C., 2013. Embree ray tracing kernels for CPUs and the Xeon Phi architecture. *International conference on computer graphics and interactive techniques, SIGGRAPH 2013, talks proceedings* [Online], 21–25 July 2013 Anaheim, California, USA. ACM, 44:1. Available from: http://dx.doi.org/10.1145/2504459.2504515.

Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J.A., Roune, B., Springer, R., Weng, X. and Hundt, R., 2016. GPUCC: An open-source GPGPU compiler. *Proceedings of the 2016 international symposium on code generation and optimization, CGO 2016* [Online], 12–18 March 2016 Barcelona, Spain, pp.105–116. Available from: http://dx.doi.org/10.1145/2854038.2854041.

Wuytack, S., Catthoor, F., Franssen, F., Nachtergaele, L. and De Man, H., 1994. Global communication and memory optimizing transformations for low power systems. *International workshop on low power design*. Vol. 203, p.208.

Yan, Y., Grossman, M. and Sarkar, V., 2009. JCUDA: A programmer-friendly interface for accelerating java programs with CUDA. In: H.J. Sips, D.H.J. Epema and H. Lin, eds. *Euro-par 2009 parallel processing, 15th international euro-par conference. proceedings* [Online], 25–29 August 2009 Delft, The Netherlands. Vol. 5704, Lecture Notes in Computer Science. Springer, pp.887–899. Available from: http://dx.doi.org/10.1007/978-3-642-03869-3_82.

Yang, C.-C., Wang, S.-C., Chen, C.-C. and Lee, J.-K., 2015. The support of an experimental OpenCL compiler on HSA environments. *Proceedings of the 21st international conference on parallel and distributed processing techniques and applications (PDPTA)*. Las Vega, NV, USA, pp.184–190.

Young, I.T., Gerbrands, J.J. and Van Vliet, L.J., 1998. *Fundamentals of image processing*. The Netherlands: Delft University of Technology.

Yuffe, M., Knoll, E., Mehalel, M., Shor, J. and Kurts, T., 2011. A fully integrated multi-CPU, GPU and memory controller 32nm processor. *IEEE international solid-state circuits conference, ISSCC 2011, digest of technical papers* [Online], 20–24 February 2011 San Francisco, CA, USA. IEEE, pp.264–266. Available from: http://dx.doi.org/10.1109/ISSCC.2011.5746311.

Zhou, K., Hou, Q., Wang, R. and Guo, B., 2008. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.* [Online], 27(5), 126:1–126:11. Available from: http://dx.doi.org/10.1145/1457515.1409079.

Žužek, P., 2016. *Implementation of the SYCL heterogeneous computing library*. MA thesis. Slovenia: University of Ljubljana.

# Part V

# Appendices

# A PUBLICATIONS AND PRESENTATIONS

## A.1 PUBLICATIONS

Portions of the work contained in this thesis have previously been described in the following publications:

Potter, R., Bradford, R.J., Murray, A. and Dolinsky, U., 2016. A C++ programming model for Heterogeneous System Architecture. In: M. Taufer, B. Mohr and J.M. Kunkel, eds. *High performance computing - ISC high performance 2016 international workshops, revised selected papers* [Online], Frankfurt, Germany. Vol. 9945, Lecture Notes in Computer Science, pp.433–450. Available from: `http://dx.doi.org/10.1007/978-3-319-46079-6_31`.

Potter, R., Keir, P., Bradford, R.J. and Murray, A., 2015. Kernel composition in SYCL. In: S. McIntosh-Smith and B. Bergen, eds. *Proceedings of the 3rd international workshop on OpenCL, IWOCL 2015* [Online], 13–13 May 2015 Palo Alto, California, USA. ACM, 11:1–11:7. Available from: `http://dx.doi.org/10.1145/2791321.2791332`.

## A.2 STANDARDS AND SPECIFICATIONS

HSA Foundation, 2015. *HSA runtime programmers reference manual* [Online]. (V.1.0). Available from: `http://www.hsafoundation.com/?ddownload=4946`.

Khronos OpenCL Working Group – SYCL subgroup, 2015. *SYCL specification* [Online]. L. Howes and M. Rovatsou, eds. (V.1.2). Khronos Group. Available from: `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf`.

## A.3 PRESENTATIONS

Lomüller, V., Potter, R. and Dolinsky, U., 2016. *C++ on accelerators: supporting single-source SYCL and HSA programming models using clang*. European LLVM Developers' Meeting. Barcelona, Spain.

Potter, R., 2014a. *Raytracing on Heterogenous System Architecture*. Glasgow Parallelism Group. Glasgow, UK.

Potter, R., 2014b. *SPIR 2.0 provisional*. OpenCL BoF, SIGGRAPH. Vancouver, Canada.

Potter, R., 2014c. *SYCL: A system which integrates C++ with OpenCL*. UK Many-Core Developer Conference. Cambridge, UK.

Potter, R., 2015a. *Implementing khronos SYCL for OpenCL*. LPGPU Workshop on Power-Efficient GPU and Many-core Computing. Amsterdam, The Netherlands.

Potter, R., 2015b. *Models for high level languages on HSA*. HSA Foundation Working Group. Cambridge, UK.

Potter, R., 2015c. *SYCL overview*. Khronos Group - OpenCL, SYCL & SPIR-V, SIGGRAPH. Los Angeles, US.

Potter, R., 2016a. *A C++ programming model for Heterogeneous System Architecture*. SG14 - ISO C++ Study Group on Games Development and Low-Latency.

Potter, R., 2016b. *A C++ programming model for Heterogeneous System Architecture*. UK Many-Core Developer Conference. Edinburgh, UK.

Potter, R., 2016c. *A C++ programming model for Heterogeneous System Architecture*. First International Workshop on Performance Portable Programming Models for Accelerators. Frankfurt, Germany.

Potter, R., 2016d. *A C++ programming model for Heterogeneous System Architecture*. HSA Foundation Working Group. Edinburgh, UK.

Potter, R., 2016e. *Generating efficient GPU kernels using expression templates*. HSA Foundation Working Group. Edinburgh, UK.

Potter, R., 2016f. *SPIR-V: A shader IR for OpenCL, OpenGL and Vulkan*. Khronos Group - OpenCL, SYCL & SPIR-V, SIGGRAPH. Anaheim, USA.

Potter, R. and Keir, P., 2013. *Fusing gpu kernels within a novel single-source c++ api* [Online]. Glasgow Parallelism Group. Glasgow, UK. Available from: http://www.dcs.gla.ac.uk/research/gpg/2013-2014-talks.htm.

Potter, R., Keir, P., Lucas, J., Alvarez-Mesa, M., Juurlink, B. and Richards, A., 2013. *Fusing GPU kernels within a novel single-source C++ API* [Online]. HiPEAC Compiler, Architecture and Tools Conference. Haifa, Israel. Available from: https://software.intel.com/en-us/articles/compiler-architecture-and-tools-conference-2013-abstract.

Potter, R., Keir, P., Lucas, J., Alvarez-Mesa, M., Juurlink, B. and Richards, A., 2014. *Fusing GPU kernels within a novel single-source C++ API* [Online]. LPGPU Workshop on Power-Efficient GPU and Many-core Computing. Vienna, Austria. Available from: http://lpgpu.org/wp/pegpum-2017/pegpum-2014/.